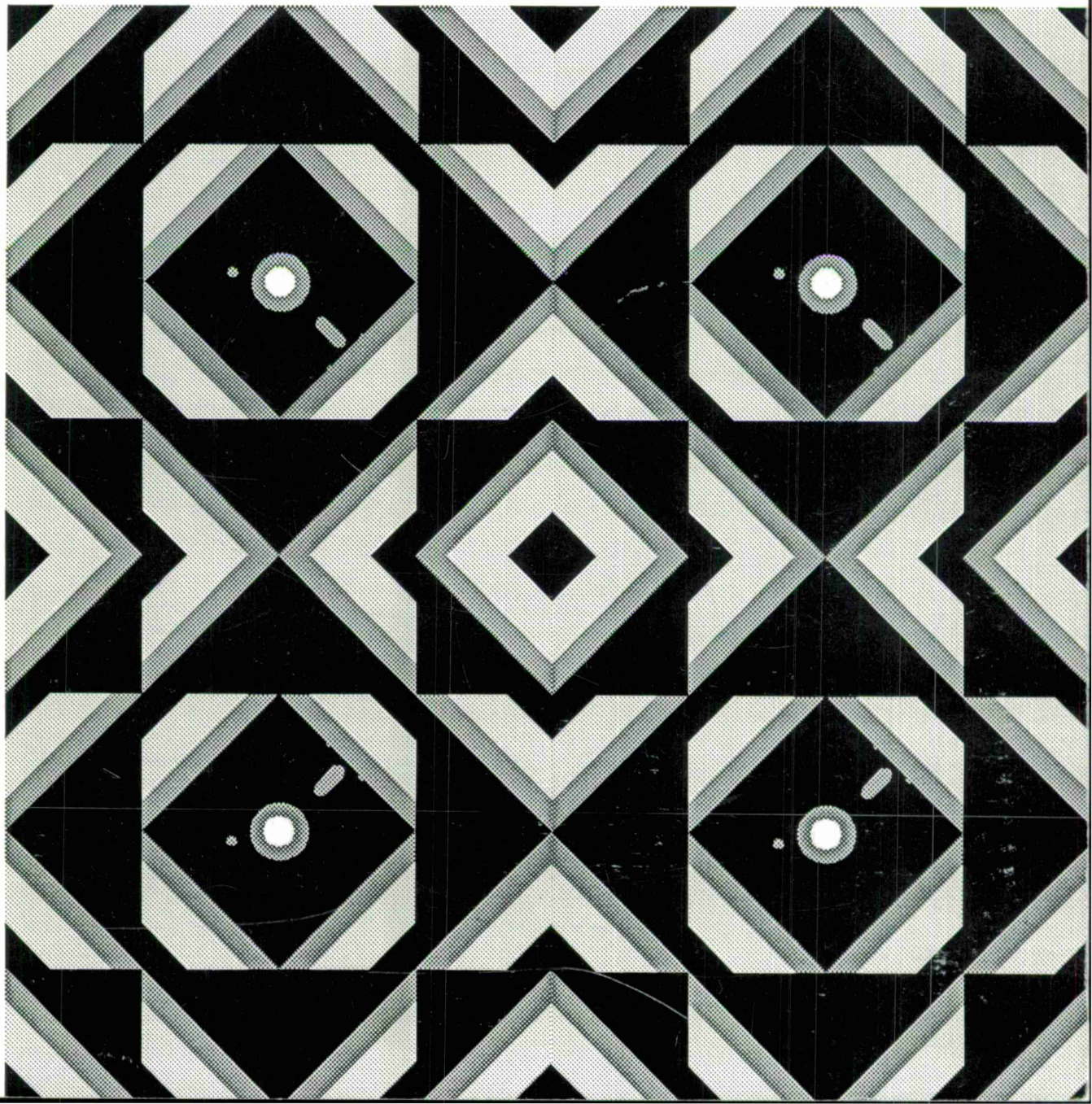
ESD/MITRE

Software Acquisition SYMPOSIUM

An ESD/Industry Dialogue

Proceedings
May 6–7, 1986



UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release, distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) M86-55 ESD-TR-87-133	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION The MITRE Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Burlington Road Bedford, MA 01730	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Deputy Commander for Development Plans (cont.) 8b. OFFICE SYMBOL (If applicable) XRS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-86-C-0001
8c. ADDRESS (City, State, and ZIP Code) Electronic Systems Division, AFSC Hanscom AFB, MA 01731-5000	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. 5720 / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
ESD/MITRE SOFTWARE ACQUISITION SYMPOSIUM PROCEEDINGS, May 6-7, 1986

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM _____ TO _____	1986 May 6-7	119

16. SUPPLEMENTARY NOTATION
This symposium was jointly sponsored by MITRE and ESD, under contract F19628-86-C-0001.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Ada*
			Software Acquisition
			Software Requirements Definition

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This document is the proceedings of the Software Acquisition Symposium held May 6 and 7, 1986 at The MITRE Corporation in Bedford, Massachusetts. The symposium was co-sponsored by MITRE and ESD. Speakers from The MITRE Corporation, the United States Air Force, and industry exchanged views on software acquisition, Ada*, and software development environments.

*Ada is a Registered Trademark of the Department of Defense (Ada Joint Program Office).

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Diana F. Arimento	22b. TELEPHONE (Include Area Code) (617)271-7454 22c. OFFICE SYMBOL Mail Stop D230

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

8a. and Support Systems.

ESD/MITRE Software Acquisition Symposium

An ESD/Industry Dialogue

May 6 and 7, 1986

Sponsors:
Electronic Systems Division, AFSC
Hanscom Air Force Base, Massachusetts

The MITRE Corporation
Bedford, Massachusetts

Symposium Chairpersons:
Robert J. Kent
Director of Computer Systems Engineering
ESD

Judith A. Clapp
Assistant Director for Software Technology
The MITRE Corporation

Approved for public release; distribution unlimited
MITRE Document M86-55

Contents

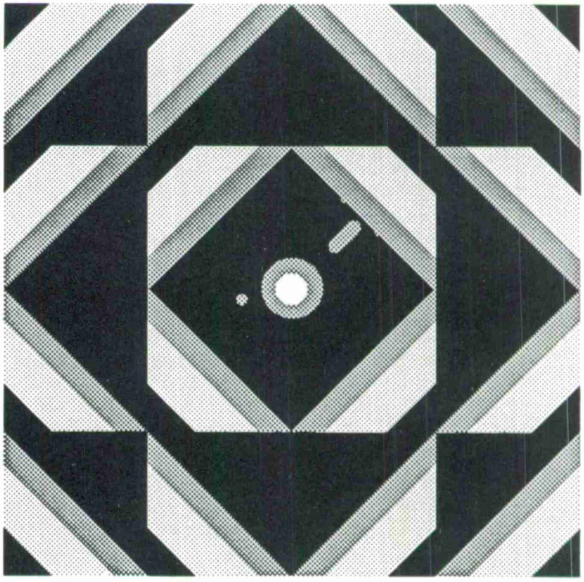
*Ada is a Registered Trademark of the Department of Defense (Ada Joint Program Office)

Session 6 Are New Business Practices Needed?

Session 7 Where Do We Go From Here?

Speaker Biographies

Opening Remarks



Charles A. Zraket
President and Chief Executive Officer
The MITRE Corporation

Good morning. I'm pleased to welcome all of you here this morning on behalf of ESD and MITRE. I would like to give you my impressions of the software acquisition problems that I have come across through our work here for ESD and over the past year or so while I have been serving on the Defense Science Board Software Task Force.

A few years ago, we conducted a major study at ESD and found that software acquisition was probably the largest acquisition problem that ESD had. It is no surprise that the continuing growth in large scale, complex, software-intensive DOD systems coupled with rapidly changing technology has strained the ability of almost every agency in the DOD to effectively manage a controlled software development. And because software drives the overall performance of both C^3I systems and weapons systems, deficiencies in software development and production have an adverse effect on overall system performance. We find here at ESD, for example, that even when software is only five to ten percent of the total acquisition cost, it essentially drives the schedules and the performance of the overall system.

This problem is exacerbated by a parallel and continuing growth of software in the commercial sector. The commercial software market is almost 20 times as large as the DOD market, so there is great competition for highly qualified people in the software business, which adds to the problem.

As all of you from industry well know, the acquisition of software-intensive DOD systems is a heavily regulated process, as exemplified by DOD Directive 5000.29, by DOD-STD-2167, and by the Federal Acquisition Regulations for Rights and Data. The rights and data problem is one that I had not appreciated until recently. The lack of clarity about industrial rights and data in software has kept DOD from being able to fully use commercial practices in software development acquisition. We on the DOD software task force have been trying to determine how we can solve this problem by changing the DOD regulatory structure. We would like to see companies benefit more from investments they might make in new software technologies that can help to increase productivity and reduce costs.

Another problem is the difficulty military users have in completely and accurately describing the operational requirements of mission-critical systems. Formulating detailed specifications seems to require a lot of iteration, and testing by real operators in an operational environment. As most of you know, this need for trial and error has led to the concept of rapid prototyping. The use of streamlined prototypes to refine requirements and define functional increments with the users will go a long way toward clearing up the software requirements problem. Our study found that the biggest bottleneck in keeping control of software development was that requirements problem.

Finally, there is the whole area of software quality. I believe that the DOD does not usually receive software products that are truly high in quality, meaning well-built, reliable software that is well-specified and documented and neatly packaged and modularized. We always seem to

go through a lot of compromises during the acquisition cycle, and the quality of the software usually suffers. We should be able to do a lot better in that respect.

In effect, then, what we are going to try to do is to address these four areas in software acquisition, looking first at the regulatory structure within the DOD to see what kind of economic incentives could be provided to contractors and what kind of incentives would increase productivity and quality of DOD software development, production, and maintenance. Then we will look at acquisition practices in risk management, rapid prototyping, reuse of software, evolutionary development, and use of better metrics, both to measure progress and to measure quality.

That is a very large agenda, but I hope that we will be able to discuss many of these issues at the meeting over the next couple of days.

Maj. Gen. Thomas C. Brandt

Vice Commander
Electronic Systems Division

The challenge that we face today is really one of demographics. I am not convinced that we have enough software engineering knowledge to propel the technology forward. To achieve success we must be more efficient and we must work hard. We at ESD are very serious about this challenge. The solutions to software development problems require a joint effort between ESD and industry.

Let me review some of our activities in this field. We have established the ESD/MITRE Software Center to concentrate on these problems and improve acquisitions. We have developed and are using software management indicators called metrics to assess the planning process and the parameters of the system. We are asking the question: "Can this be completed on time and at that cost?" We are using "red teams," made up of software experts, to resolve the problems that stand in the way of efficient and effective acquisition. We are focusing on the Air Force's Computer Resources Management Technology Program to accelerate the implementation of new software acquisitions and the insertion of these technologies. Finally, we are showcasing all of this in the new DOD Software Engineering Institute at Carnegie-Mellon University in Pittsburgh.

There are a lot of initiatives underway, and a symposium is a wonderful forum for exchanging ideas. It permits us to ask questions such as, "Am I right or wrong?" and, "If I am right, how are we going to solve the problem?" We have to establish a larger body of software expertise. The projected growth in the need for software appears to be exponential. That is a tremendous challenge.

I believe we ought to consider the use of a lot more engineering before full scale development. I keep asking the folks around here, "Where are my brassboards, my testbeds; where is my Exploratory Development money?" And they say, "We don't have any of that, we are going to full-scale engineering development." And that, of course, implies an assumption that risk is low. Now since I have tended to be a purist throughout my life, how can you say risk is low if two out of three programs are behind schedule or the cost is doubled? We are kidding ourselves. If that is the case, the risk was not low, whether someone stood up and said it was or not.

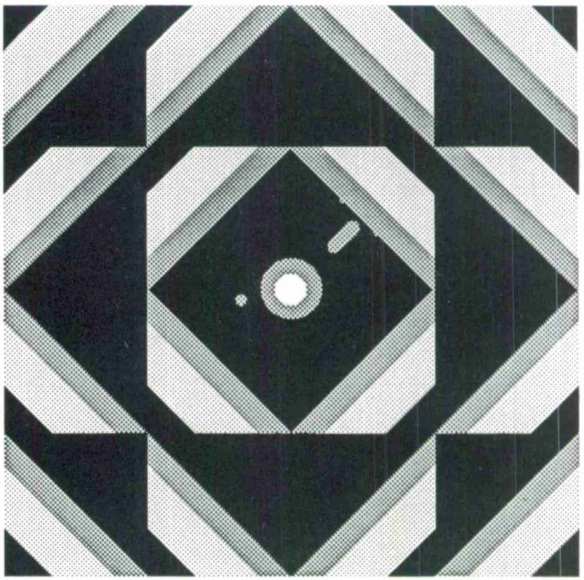
Look at this year in the space business. Half a dozen years ago I participated with the Scientific Advisory Board in a summer study on Space, and we recognized the need for a mixed fleet to reliably lift our military space satellites. The shuttle is a wonderful machine, but it was wrong to assume that it would work every time, forever. That is crazy, yet it became a fundamental assumption underpinning all our decisions in our national space policy. Bad logic.

Now we are looking at the Titan 34D rocket as an alternative, and it has failed a couple of times. The Delta rocket failed recently. We have not entirely mastered this complex technology.

There is a tremendous amount of hard work ahead, and we have got to be rational and realistic to meet the challenges. One of the biggest of those challenges for the scientific and engineering community is to get our acts together and improve the software development process.

Session 1

ESD/MITRE Views of Software Acquisition



Moderator: Judith A. Clapp

Anthony D. Salvucci

Assistant Deputy Commander for Strategic Systems
Electronic Systems Division

In my first year and a half at ESD, I was part of the operational world with the responsibility for tracking satellites and computing their orbits. In those days, there weren't many satellites. There were four analysts, including myself, and we each computed the orbit for our own satellite. We used desk calculators for this job. It was a rather crude form of computation, but it was reasonable enough since there wasn't much we could do with the information other than discuss it with astronomers who would say I guess you guys were right, the satellite was there in approximately the time and about the right quadrant of the sky as you predicted. Today, that process has changed drastically, as strategic C^3 business has grown considerably over the years.

Strategic C^3 is a process of trying to improve strategic connectivity, which has three basic mission areas:

- To be able to provide warning of attack, whether it's atmospheric, missile, or from space.
- To be able to get that information to the command centers where decisions are made.
- To control the forces. This, of course, requires a variety of communications media that can last through all forms of conflict.

One could argue that communications is the glue that binds the strategic world together. Another view is that strategic connectivity is made possible by software. Software has become the cornerstone of each and every element of the strategic defense posture. And much of this software must be developed along with a strategic program. Very little off-the-shelf software has been available to us. This is one challenge for the software business.

Software development problems are as varied and widespread as are strategic programs, but government executives must attempt to solve them. I would like to address these problems from my perspective as a manager of strategic systems by discussing what I see as some of the causes.

First of all, software development in the Department of Defense has gotten larger and our dependence on software has grown enormously. When we look across the world as a whole, we see that information management in systems, of which data processing systems and software are essentially the foundation, is booming. It is the largest developmental area that we have.

As the expansion continues, we need an increasing number of people to accomplish the task. Our current personnel resources are being stretched thinner and thinner. Today, the demand for people is greater than the supply. An initial analysis done by a number of people, including our Software Engineering Institute, indicates we are graduating fewer and fewer people in this business. The results show students coming out of the elementary and the high schools lack interest in the sciences in general and in computer sciences in particular. We are all clamoring for these few resources. This is challenge in and of itself.

Not too long ago, I surveyed the major companies we do business with and found the statistics were essentially the same for all companies. The so-called experienced systems engineers design-

ing our software had an average of five years of experience. Most of the staff had less than five years of experience, and the old timers were people with five to eight years of experience. I think those numbers could easily have been twice as high 20 years ago. Managers would simply pull the best people together for a critical development.

Today, there just aren't enough "best people." The problem is magnified by the fact that the challenge is getting bigger and technology is promising more, but the people to acquire it are becoming more scarce. This dilemma is common to all the programs. The human resources problem cannot be solved overnight.

I am also concerned by the fact that we don't seem to get the best quality software teams to work in the Department of Defense (DOD) business. There are a number of arguments as to why. I don't want to go into them all because I guess I'm not certain as to the underlying cause, and there may be more than one. The data rights problems are probably one cause. Rights protection is not afforded to contractors who have developed software with their own investment. Innovative developers, as a result, don't want to work with the DOD. Others believe the problem is the way we do business in the DOD, and not a data rights problem. They point out that the DOD develops the capabilities it needs rather than motivates people in the private sector to solve the problems and have the solutions available for us to purchase. But the bottom line is that we are not getting the best people to work on our programs.

Another concern I have is the use of what ESD calls "graybeard teams." When we are in the process of selecting a contractor, we receive a great deal of effort in the proposal phase of the program. The best technical writers are brought together to present the best cases in order to win the job.

We often find, however, that teams mustered together to work the program from day one until it's completed aren't necessarily of the same caliber. The selection process, therefore, is not based on picking the best team for the job, but the best company presentation. We have even found companies who hire other companies to write proposals. This doesn't help us choose the right team to work the problem. Since qualified people are crucial to the development process, this is an important aspect of the selection process.

The difficulty with the qualified people is not just on the contractor's side. We have a similar problem on the government side. There are too few experienced people to go around. As a result, we often find out after the fact that pivotal decisions in a program are made by newcomers, by those who are working hard, who have got a lot of potential, but don't have much experience to bring to the floor. As a result, many selections are best choices based on limited experience of the teams formed to evaluate the bids.

We have turned to a process in which we take the battle-scarred members of our staff, so-called "graybeards," for critical procurements and perform what I call "orals" with the contractors. I'm not as interested in a proposal as I am in who is going to work the program, who the chief engineer is, and who is going to run the software. It makes sense to identify the key players and discuss with them the pending procurement, their approach to the problem, their experiences, and the "what-ifs" in terms of how they have planned their activities for those times when everything does not go well.

We get some interesting results in this process that don't always match up with the evaluated results from the paper proposals delivered by the contractors. The process we're using is valuable.

I'd like to say a few words on management by metrics. We are supporting the application of metrics to software management. Most people address metrics as tools for inspecting the pro-

cess, and they can certainly be used for that. One of the greatest advantages of metrics, however, is that they exist in a formal way, that there is a plan. Too often we find that activities, both on the government side and the contractor side, get off and running before there is a plan of what is to be accomplished. We must determine ahead of time how we expect to get to the goal, and what resource will be used. Too often we make selections based on the good ideas people have without worrying about the mechanics of getting there.

A primary value of software metrics is that they support this kind of planning. Taking snapshots or slices of a program and doing that for each element of the program is a very good way of checking on the progress of a plan. It can help make sure that all the resources allocated to the job are in fact being used, and it can verify whether you're going to get to the end of the product on a reasonable time match as outlined in the contract.

The inspection process is also important. In the inspection process, there are two ways of looking at the data that would fill in a metrics chart. One, of course, shows where you have been which gives you an idea of how well you are doing.

Another value of software metrics is that they can be used to compare your forecast to the forecast that your program manager or software manager is giving you (Figure 1). The curve provided by the metrics represents the productivity of your software team. If you have gone to many program reviews, you always find program managers — and I was one — who tell you that, no sweat, just a little bend in that curve and we will get there. But if that little bend represents double the productivity over that phase of the program, is that possible? Even if I were willing to double the amount of the resources, is the experience base of the resources I might add equivalent to the experience of the team that has built up that productivity curve? Software metrics

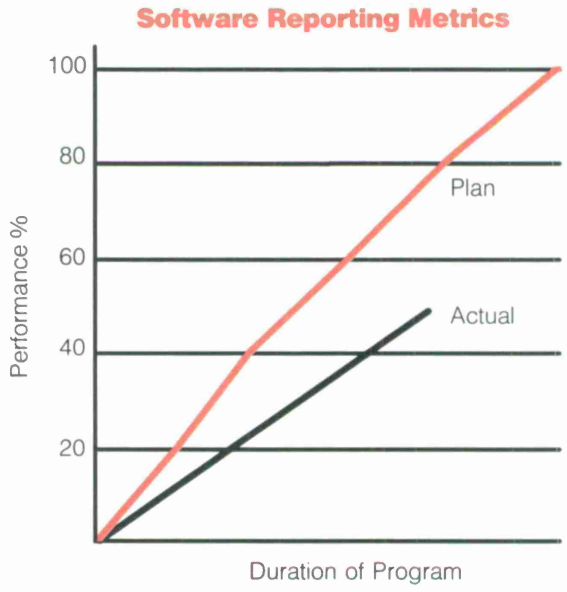


Figure 1

can be very valuable if they stimulate discussions about what is really possible at a given point in the program. They can prompt one to think in quantitative terms about where the program really is and where it might go.

The subject of internal contractual audits is not a popular topic, but I am convinced that we have missed an opportunity in this area. We have built up over the last several years an interest, or perhaps a need, to form Independent Validation and Verification (IV&V) teams. These teams are independent in the sense that the government will form a team or hire a company to do that work for them and monitor the work of the contractors. Having been inspected in the past, I understand the feelings of those being inspected, but being at management level in this business today I also recognize the need for inspection.

I believe that we have used the IV&V approach incorrectly. It hasn't solved the problem because management on the government side is not going to work the problem. If the problem has been

contracted out to industry, it's management on the industry side that needs to work the problem. Those are the people that are being paid to work on the problem.

I don't see enough formalized internal inspections by industry management. That is what I mean by that internal contractor audit. I do not mean government inspection teams; they should be contractor independent inspection teams that ultimately report to the same boss somewhere in the hierarchy of that corporation.

Any of you in the DOD that have been in an Operational Command and have faced up to an Operational Readiness Inspection (ORI), know that you give the inspection team a lot of respect and they get access to all the data they need. When they report something, they go overboard with regard to identifying the details and the facts that are there and, of course, their own conclusions on the causes of the problem and potential solutions. Management in industry needs that kind of insight in order to make decisions.

Not too long ago, General Bill Creech liked to push the philosophy that you always keep management informed as soon as a problem occurs. We don't see enough of that soon enough in industry. Problems are usually identified too late for management to act, and sometimes only as a result of government pointing out what the problem is. When the government points out what the problem is, the contractor at some level is usually aware of it, but not at a high enough level to be effective at controlling it.

The future holds a number of challenges. The problem of insufficient personnel resources is the most fundamental one. We must somehow free ourselves from dependence on labor resources to accomplish our tasks. The problem of supply of labor is compounded by the growth of our business. We must turn to technology to assist us as we are trying to do in every other arena to lessen our overall dependence on a large, highly skilled labor force.

Richard J. Sylvester
Director, MITRE Software Center
The MITRE Corporation

I want to talk to you today about what I believe to be a change of emphasis in the last year and a half at ESD. General Chubb has instituted a number of ad hoc government assessment teams to review the status (primarily the software status) of various programs. There have been about twelve of these "red team" reviews focusing on software. Some of you are from companies that have had the "pleasure" of our visits. I have been affiliated with six of these twelve reviews during that year and a half.

The purpose of the government assessment teams or "red teams" is to improve communications about what is happening on a specific program and to come up with assessments and recommendations. These teams are usually structured with some Air Force personnel and some MITRE personnel. A team may consist of one or two people for a very quick assessment, or as many as ten people with outside consultants in special areas, in which the assessment may take a month or longer.

The team usually prepares a charter that focuses their activities. It may be instituted during any phase of a program — during source selection, in some cases or prior to Preliminary Design Review (PDR) — but generally the team is instituted later in the program, when symptoms of problems may be visible, often during integration and testing. The team usually visits contractors and subcontractors involved with the development, and the test and evaluation agencies, if the program is that far along. For background information, the team visits the user as well as people in the Air Force program office and in the project offices at MITRE.

Typically, a good review will focus as much on documentation — existing hard evidence — as it will on conversations with people at the various locations. These assessment efforts are very, very intensive. The idea is to get the most accurate picture possible of the status of the project, so that a reasonably good set of judgments and recommendations can be made.

Industry seems to have picked up on this idea. There have been a number of industry red teams that check their own projects more or less independently. I think that is a very good idea. It gives the project people an independent assessment of what they are doing in industry, and I expect to see more of that to come.

I would like to show you the results from three such case studies; then I will discuss eight general areas in which problems tend to exist. Case 1 is a digital communications system; Case 2 is a large radar system; and Case 3 is a command and control center.

Let's begin with some characteristics of Case 1, a digital communications system. There are about 150 nodes in the system, one node being a central node with a great deal more traffic. The software for this system comprises 250,000 lines of assembly language code and three major categorizations of software. One category dealt with what is called base software, which is a secure element of the system, one dealt with interface software, and the third dealt with application software. There were two subcontractors and one prime contractor involved. One of the subcontractors did the bulk of the base software, another subcontractor did the bulk of the application software, and the prime contractor integrated the system.

This particular system experienced initial operational test and evaluation (IOT&E), and as a result, software instabilities were identified. In other words, during IOT&E, the test spanned a scope larger than the specification for the software indicated. "Creative play time," where operators were trying to break the system, was involved; when that happened, the system as it was designed shut down and restarted itself. There were losses of throughput during the operation of the system. Certain high priority messages which were required to be delivered at certain times were delayed. There was a peak loading to be achieved; it was not quite achieved during the initial operational tests. In all, some 360 software deficiencies were reported over the month of the IOT&E.

The assessment team's approach was to visit all the concerned parties and to gather as much documented evidence as possible. The team looked at computer program development plans, specifications, informal work, trouble reports, and schedules. The assessment was really beneficial to this program because the team was able to show that the program in general was not a software disaster, that the trends were positive, and more time was needed.

The first kind of trending information that the red team generated was error-free performance (Figure 1). On the horizontal axis, seven months of schedule and time are shown, and below that the amount of achieved peak load is shown — peak minute load that was achievable during a specific scenario. On the vertical axis are the error-free hours that the system performed under load. These data were put together by the team by taking individual 30-minute tests with a peak minute load during the middle of the test. Those tests were concatenated into 10-hour averages and the trends were drawn up. Clearly, there was an improvement in the trend of error-free performance over the seven-month period. That gave the team an indication that the software in the system would eventually be able to operate under peak minute load and be stable.

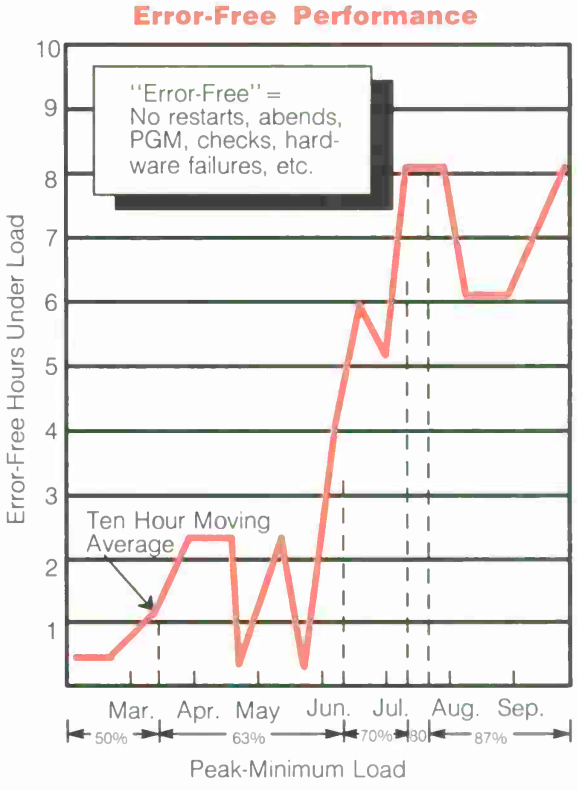


Figure 1

A second bit of trending information dealt with the software Program Trouble Reports (PTRs) and their closure histories over a long period of time (Figure 2). Here, the three major software categories plus the total operational software are shown. At the time, those curves seemed to be approaching a horizontal line. There was reason to believe that the system was improving. The team looked in more detail at the outstanding trouble reports at the time of the review. On that basis, they were able to determine that the trouble reports were localized in nature, that correcting those troubles impacted only parts of the software. The contractors and subcontractors had good trouble report systems, so the team could categorize the trouble reports and determine that the problems were not global in nature.

The team concluded that the software didn't have to be extensively rewritten in order to make the system work properly. The fixes were localized. However, the operational tests showed that there was a design philosophy in handling operator input errors that was not appropriate; and even though the specifications may have been satisfied, the operational testing extending beyond the specification hadn't been satisfied.

There was an issue of retaining key personnel among the subcontractors. These people were scheduled to come off the program much too early; they were absolutely necessary to handle the cited deficiencies. The documentation exhibited a great leap from requirements to design. The key personnel had the information needed to make that leap in their minds but not on paper, so it was necessary for them to stay with the project either until the problems were resolved or until the information they had was documented well enough that someone else could resolve the problems.

The trends in PTR closures and load handling were very encouraging. However, the teams felt that the schedule for the completion of the corrections was very optimistic, and that it was more success-oriented than necessary. The team felt that the process to make the corrections and have the system operationally tested a second time could fail if the work was not event-oriented rather than schedule-oriented. The team tried to allocate enough schedule time and resources in their recommendations, so that this particular project had a good chance of satisfying a second test and evaluation activity.

The second case history is a large radar system. Clearly, there were some important real-time response requirements. One prime contractor and two subcontractors were involved. The red team had been set up to check why milestones were missed. There was a particular set of display problems, the solutions of which were important to the success of this project.

The red team in this activity was looking at a program that was two-thirds complete. The following criteria should have been met at this point in the development cycle: requirements and design complete and stable, particularly software design; baseline strongly managed; interfaces defined and controlled; coding nearly complete; test and diagnostic tools in place; a strong integration plan; and the remaining activities scheduled with adequate reserves.

Cumulative Software Program Trouble Report Closure

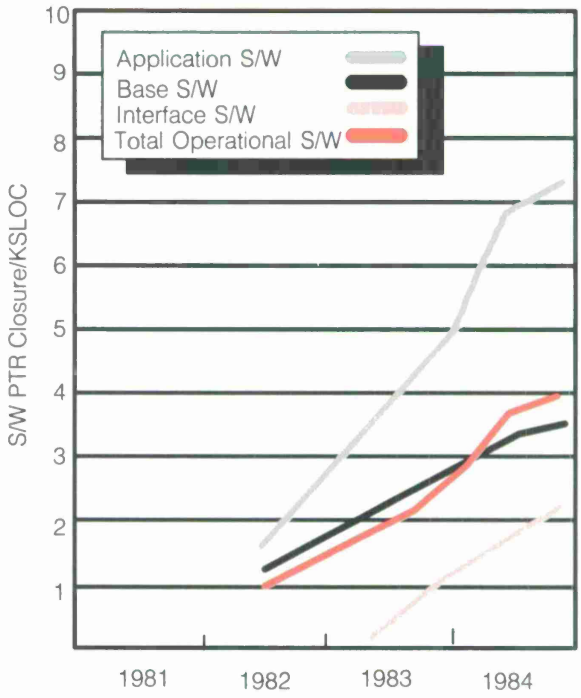


Figure 2

Now let's see what the team found. Many activities were being run in parallel and were planned that way. Individual activities had highly optimistic schedules. Because of the highly parallel schedule, the effects of slips tended to ripple through the schedule and serialize it because the parallel activities required more coordination than the team thought was possible. They felt that the troubles would be pushed downstream to meet short-term schedules, making software

integration even more difficult. People would take the easy things first, make the token milestones, and push troubles downstream; the schedule problems would get worse, and any kind of regression testing would probably be limited or would ripple through the whole schedule causing further extensions.

The most critical problem in the team's view was the display systems. The displays were the principal tools for doing the software test and the integration test, yet they were the most complicated elements because there was hardware from three different vendors and software being done by the prime contractor and the two subcontractors. Any delay or inoperability of the display system would seriously impact the integration testing.

There were several problems with the display system. There was a "hang" problem; with certain communications to the display, the display would stop the entire system, and the system would have to be restarted. There was a response time problem. The displays were responding to operator actions in a much longer time than the specification indicated. These were critical problems.

Not all of the display software was being done to the same baseline. The subcontractor, who was on a firm fixed-price subcontract, was working toward the original baseline. The prime contractor had upgraded his own baseline, but had not upgraded the subcontractor's baseline.

There were a variety of other problems, not the least of which was that the management was not really paying very much attention to the software problems. In subsequent months, the display problems did slow down the integration and testing substantially and impacted the schedule, as was anticipated. A series of integration difficulties were also identified.

At this point of the program, after 32 months out of 48, one would expect the design to be pretty stable, since software was being written. However, there were elements of the design that were not stable. There were questions about what the design meant to the people who were implementing it. There was a process for answering these questions, but it impacted the schedule. Some maturity in the design at an earlier point would have been the right approach in this activity.

Let me move on to Case 3, the Command and Control Center. The software consisted of 250,000 lines of FORTRAN and assembly code, and was to be run with a commercial operating system. There were two subcontractors and one prime contractor on the job. This program had a variety of schedule problems. I will focus on the test and integration phase because this is where everything seems to hinge. When one starts losing the schedule up front and one holds to the same completion date, the result is a compression of test and integration.

In this example, there were three estimates of the schedule (The earlier part of the program before coding is not shown on the schedule in Figure 3). At the time of award, the schedule showed nine months to accomplish the coding until the start of formal qualification tests. Formal Qualification Test (FQT) was three months in length. The integration test was four months in length, with an Initial Operational Test and Evaluation (IOT&E) of about a month, then an installation phase, ready for Initial Operational Capability (IOC).

A second schedule estimate was made by the contractor in December of the year 198A, and we see a slip at the front end. The actual start was three months later. We had a stretch of a month in the FQT, and a stretch of a month in the integration of tests.

Four months later in March of 198B, a new update to the schedule showed that coding prior to FQT took 12 months, FQT would take seven months, and the equipment could not be installed until later because the place where it was to be installed wasn't available. So we had the initial estimate, the December estimate, and then the March estimate.

History of Schedule Changes

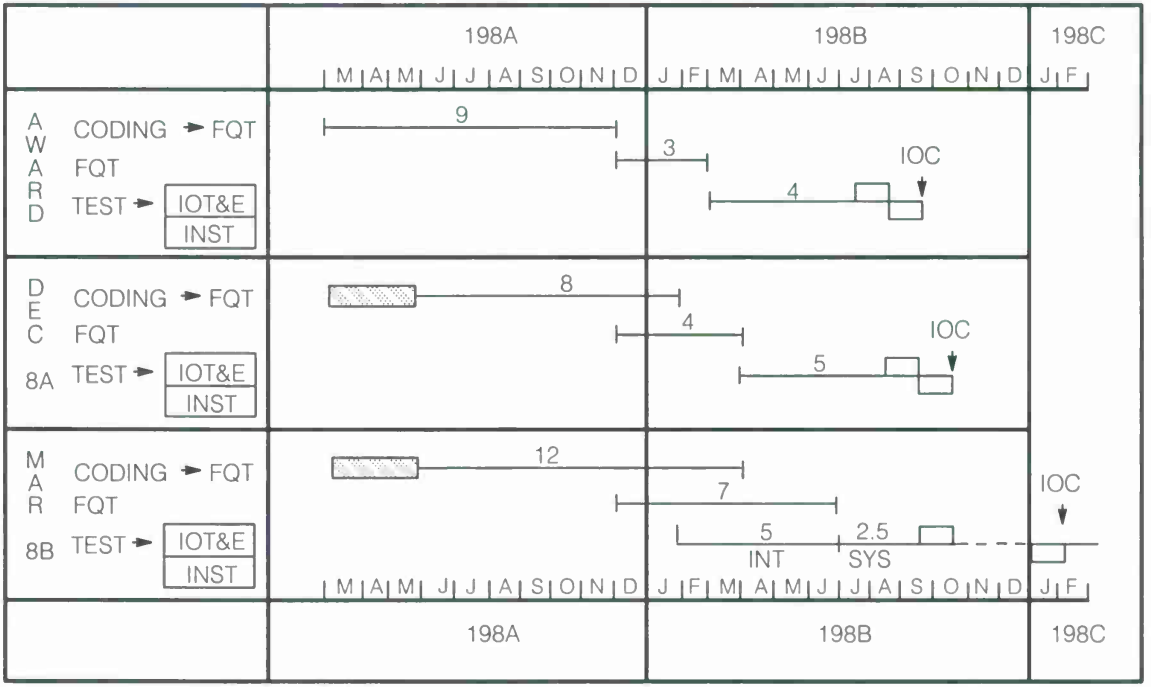


Figure 3

Let's take a look at the FQT process. Figure 4 shows the period of time from the beginning of December to the end of February for the contract award estimate for FQT. The December estimate ran from the beginning of December to the end of March. The March estimate ran from the beginning of March to the end of June. This was the status when a review was done at the end of May.

Figure 5 shows the March 198B integration test vs. actual tests. The integration testing itself is stretched out considerably farther than the plan. The contractor was far behind plan one month after this plan was made. This kind of trending information is valuable in assessing how you are doing on the schedule.

Some 353 tests had been planned, of which about 290 or 82% had been completed. Something like 7,700 discrepancy reports had been made, and there had been no slowing in the rate of the reports.

It would be very helpful to have a good, rational way to try to project the completion of problem reports based on some good, historic data that one could gather during the process. That seems to me to be very difficult to manage.

In many cases hexadecimal dumps were being used to try to debug, and this was taking a lot of time. At this point in the program, there was no idea as to whether system performance was going to be met because the software was not being integrated on the final configuration. There was data only from a model and not from the execution of the software itself in order to assess performance. By January of 198C, the software was really not considered mature.

Those are the three test cases. Let me make a few comments about the variety of common problems that the review teams have unearthed.

Often, the contractors selected are new at dealing with the Air Force or ESD, and have never done B specifications before. It's very rare that we get a good B5 specification for software; there is either too much design information that is baselined, or too little information.

Another common problem is that the definition of a stress test or loading test is often weak in the specification or left for later definition. Until the load test is defined, the goals for sizing and timing are difficult to attain and design is not finalized. Unless the load test is realistic when you get into operational tests, the system could be deemed unsatisfactory.

It is very important to capture the software design with good documentation and not leave it in the minds of the developers. Perhaps DOD-STD-2167 will help do that. Red teams see big jumps from the B5 specifications to the C5 specifications in some of the programs that they review.

As I have already discussed, if a program manager remains schedule-driven when he's in serious trouble, he tends to get into worse trouble. We have seen a number of programs where the discipline breaks down when you try to drive against an unrealistic schedule and the people can't do it. If the contractor is too tightly constrained by the schedule and cannot put qualified staff on to relieve that constraint, one has to think seriously about adopting a schedule that is event-driven.

One of the reasons test and integration are complex is that there is a tendency to define configuration items orthogonal to easy integration. I have seen several programs where the configuration items take a lot of scaffolding in order to exercise them before they can be integrated. A different definition of configuration items would allow them to be implemented end-to-end, in levels and in smaller blocks, and tested by releases. This would permit additions for more capability at another level, and could save on the building of special drivers for the testing activity.

Management attention to software is a key element. Sometimes the program manager just doesn't tune in; he lets somebody else handle it,

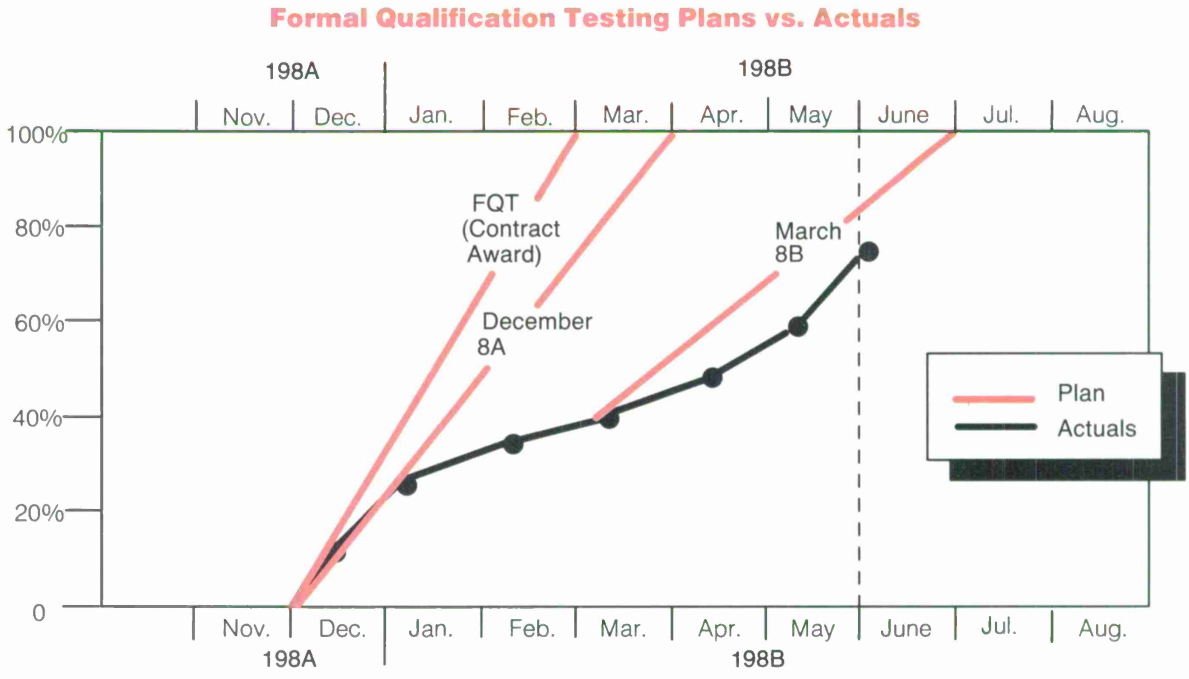


Figure 4

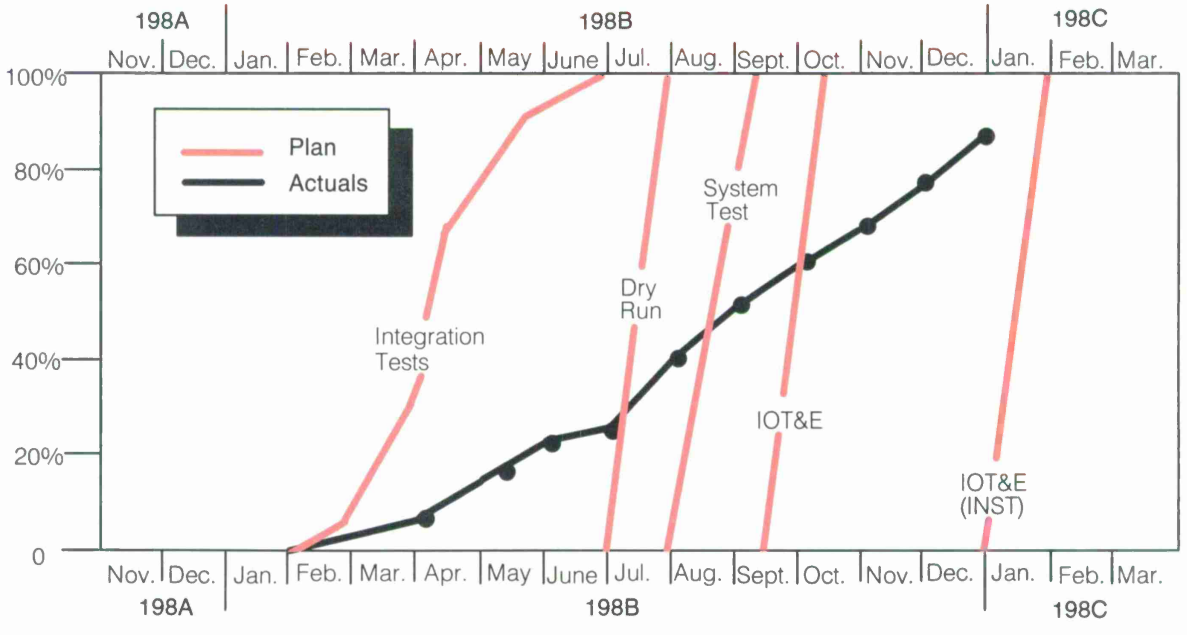


Figure 5

and he tries to make decisions based on someone else's interpretation. But, the software integrates the system; the software provides the system with its functionality. So it's important to apply the right management skills, even at the top level.

Subcontractor management is extremely important as well. Contractors need the visibility into what their subcontractor is doing. If it's a fixed-price subcontract, he's going to work right to the letter of the contract. As a prime contractor, one must have visibility into the subcontractor's work and assure proper communication, so that the subcontractor's activities integrate into the system as a whole.

Lt. Col. William E. Koss

System Program Director for Granite Sentry
Electronic Systems Division

Over the last 15 years or so, there have been many studies on the software problem. These studies were folded into initiatives that in turn are folded into bureaucracies. In most cases we don't really solve the problem, so we go back and study it again, and the process continues. I will try to bring together the initiatives and studies on software development and the actual practice of software development. Given the entire universe of activities in software development, what do we need to do, and what are the things that actually make a difference?

The software development process today is very labor intensive, and all of our major systems have a work breakdown structure. The more complex a software system is, the more we must compensate for that complexity with a longer schedule to allow the communication deficiencies to be brought into line with the software development.

What may start out as a straightforward problem becomes a complex situation when you consider the management structure, the organizational structure, and the communication inefficiencies. The automobile industry is an example: an American manufacturer has about 10 levels of management and the Japanese manufacturers have three or four. Those extra six layers of management take their toll in product quality, cost, and schedule. Communications inefficiencies are very expensive, and the ability to minimize them is the key to a cost-effective software development.

As we compensate for complexity by extending the schedule, we will bring the cost down to a certain point, but we're always fighting some directed date for an Initial Operational Capability

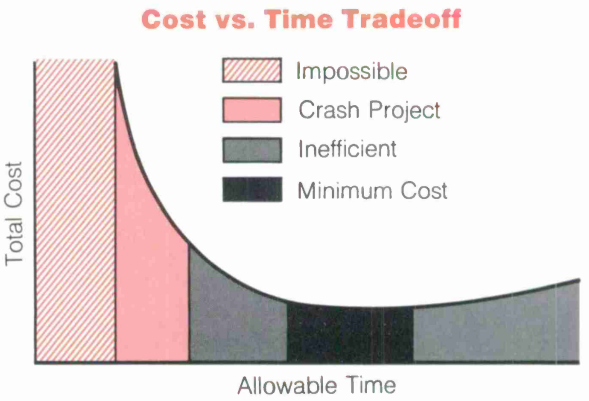


Figure 1: with limits, we can trade money for elapsed time, but it is crucial to recognize the nonlinear nature of this trade-off and the potential impact on product quality.

(IOC) that often doesn't factor in the complexities of the development. The software is vitally needed, so we compress the schedule and the cost goes up (Figure 1). When the schedule is compressed there is often a very rapid increase in cost, and this is characteristic of most defense software developments.

In 1983, we had a Space Division proposal for a very large program. We awarded the contract and directed that six months later we would have software Preliminary Design Review (PDR). This first phase of the program represented one billion dollars. To be responsive, the contractors had to fit all of that work into six months. The contractor proposed 145 parallel tasks. Even if you had one person assigned to each of those 145 tasks, it would take six months just to have each person talk with one another. We tend to

bring that type of problem upon ourselves, and it certainly points to the lack of practical software development expertise.

The key point is that we can trade money for elapsed time, but it's crucial to recognize the non-linear nature of this trade-off and the potential impact on product quality. Obviously, if you're going to compress the schedule past a certain point, the reliability of your software is certainly going to diminish and the cost will go up significantly.

This was known in 1970, yet we still haven't corrected it 16 years later. The error rates continue to go up dramatically right at the point in the development on which we spend the least time and money: those activities leading to PDR.

Over the years I have gathered data on this from industry, especially from three contractors who have done their own internal reviews to find out where the errors actually occurred in software development. A total of 85 percent of them occurred in analysis and design. Only 28 percent of those errors were found during analysis and design. The other 50 percent were found in programming test, and 22 percent of the errors were left in operations and maintenance.

I'm now in a program where I have to deal with maintenance issues. If the cost of fixing those errors in maintenance was the same as the cost of fixing them at the outset, I wouldn't worry about it, but because the cost goes up, we have to worry about detecting those errors at the right time. Those errors will not be detected at the right time in a complex program if you have a compressed schedule. People want their PDR in six months, and they're going to get it in six months, so the software will be of poor quality and will be very expensive to correct and maintain in the field.

There is a severe economic penalty for correcting those software errors in the operational phase — as much as seven to ten times the cost of correcting them during design. The cost for large complex systems can be many times more. As we proceed with more software systems on compressed schedules, it's not clear that we have enough money in the U.S. Treasury to maintain all these systems as they come on board. We really haven't fielded that much software to date compared to what we will field in the next 5 to 10 years. We must be able to maintain these systems in addition to developing them.

Software reliability presents another challenge. As a field of study, software reliability is very new, and it's not at all clear how stable that body of knowledge is. There is a vast difference between the reliability of computer hardware and the reliability of computer software. For example, a contractor estimated that the Mean Time Between Failures (MTBF) for his processors was 20,000 to 30,000 hours. The MTBF for developed software was 170 hours and the MTBF for commercial software was 670 hours.

To solve this problem, we must do a much better job of developing the B5 software development specifications. I have not seen a program yet that met cost/schedule/performance that did not have an exceptionally good B5 specification — there is a direct correlation in my experience.

In the space business, there is also a direct relationship between the precision and correctness of software requirements and the reliability of that software. Even if you don't do anything else right, you at least have to be able to create good, correct requirements. The software PDR is a single critical, credible milestone by which to assess technical requirements stability. Often, the PDR is also the only point at which we can talk about reliability and testability. We have to be able to say that we allocated to the software precisely those things that we want it to do to support the system specification.

I am firm in my belief that these crucial aspects of the program require a good B5 specification. If you pass this milestone without getting quality technical assessments, you're not going to have

any more assessments. It's going to be management by miracle because you're not going to worry anymore about assessments — you're going to worry about meeting the next milestone and hope that all the program activities produce something.

The need for a B5 specification has long been established. Whether or not we have ever met those criteria, we know what we should be doing and have adequately documented that. In most cases we have failed in this activity, and I'm saying we are failing in 1985 and 1986. We are not meeting this test. If you don't have it in the specification, then the software is simply not going to be there. I have found that the B5 reflects reality so well that if you do not pay attention to it, you won't make an IOC, regardless of your dedication.

If we know what we're doing, it will be in the B5 specification. If it's not in the B5 specification at the PDR milestone, we have to start looking at the cost and schedule, and then go back to the conceptual phase and determine when we know what we need to know. As soon as we do that, we have to capture it in the B5 specification. Most B5 specifications in no way reflect what we know. Rather, the B5 is a general outline. In the cases I have been involved with over the years, we are really only about 30 percent there at PDR. Technical reality and management reality are indifferent to directed IOCs. Reality always wins.

Delbert D. DeForest
Associate Department Head
The MITRE Corporation

In 1984, General Chubb directed the establishment of a working group to develop metrics for reporting on software development in ESD systems. A survey of people within ESD and MITRE elicited opinions on what set of parameters might help track the status of software development efforts. This culminated in a report in 1985 that was released to ESD, MITRE, and industry. The National Security Industry Association (NSIA) Task Force reviewed the report for us and gave us their comments. The second version of the report has been released and is now available through ESD and MITRE.

We have defined eight metrics, each with a different purpose. The metrics are: Software Size, Software Personnel, Software Complexity, Development Progress, Testing Progress, Computer Resource Usage, Program Volatility, and Incremental Release Content. I would like to go through the metrics and define what each one is and how we intend to use it.

Program Size is still the best way we have of determining the effort needed to develop a program. Program size is characterized by three components: new software; modified software, which is existing software that we think we can reuse by modifying it slightly; and lifted software, which is used as is.

The initial estimates of these sizes constitute a plan. It's important for us to track this throughout the development to see how the plan might change. If the estimates are not accurate, then the effort is going to change.

You cannot track only the top level. If you look only at the total, you can't really see things like the shifting that's happening among the three kinds of code — as they shift, your effort is certainly going to shift. We are always optimistic at the beginning of a program. We may expect to lift a large amount of existing code, but by the time we get to the end of the program, we find that it wasn't always possible. The less code you can lift, the higher your resource requirement. We want to watch this and make sure that the manpower adjustments are made, that the computing resource adjustments are made, and that realignment of the effort and the tracking are done as the program progresses.

Software Personnel needs to be tracked. In the past, we have only tracked the total staffing sizes; now we also need to know the number of experienced people that are applied to the program. In this case, we have a plan that is generated from whatever mechanisms are used to decide what the total effort will be; we then track that plan on a month-by-month basis. We also like to know what the unplanned attrition is. Of course, later in the program, a higher attrition rate has a greater impact.

Complexity is difficult to define. Estimates of complexity are not normally maintained throughout the development cycle, yet most costing algorithms today use a complexity factor. We are trying to encourage a reevaluation of complexity as the program progresses. This should institute a reallocation of resources if complexity starts to shift from different components or different Computer Program Configuration Items (CPCIs). We would expect to see the reallocation of resources following changes in the complexity factors. We

are not insisting on a particular complexity factor. Those that the individual contractors are using are totally acceptable; we simply want to track the changes.

Development Progress refers to the schedule of activities during the implementation phases — the detailed design, coding, and testing of the modules. We are asking for a plan for the number of units designed over time, the number of units tested over time, and the number of units integrated into Computer Software Components (CSCs), Computer Software Configuration Items (CSCIs), or CPCIs over time. We will track the actual progress against the plan over this period of time. We are looking for discrepancies so we can understand what is happening at that time and what it may mean later in the development cycle.

Testing Progress is monitored in much the same way. It is a quantitative measure, not a qualitative measure. We are looking at the number of tests planned over a period of time. This method, of course, applies during Preliminary Qualification Testing (PQT) and on into Formal Qualification Testing (FQT) and system testing. The purpose is to track progress against the plan.

Progress of testing is also associated with problem reports. We are doing some trend analysis to see what the closure rate is. We are asking for information on new reports generated over a reporting period, so that we can determine how the problem resolution process is progressing. This gives us an indication of how each of the testing efforts is progressing, and how we are meeting the schedule in completing the testing efforts.

Computer Resource Utilization refers to three components: on-line memory, input/output utilization, and CPU time. There is a budget established for most programs, so we compare the actual utilization against the budget line. This allows us to address these problems in time before we get into critical parts of development.

Program Volatility is really a measure of the stability of the requirements. With this metric, we are able to track action items, generated out of either design review meetings or technical interchange meetings, or effect of Engineering Change Proposals (ECPs), which are always generated with a line of code estimate of the impact. We track the ECP impacts as well as Advance Study Change Notices (ASCNs).

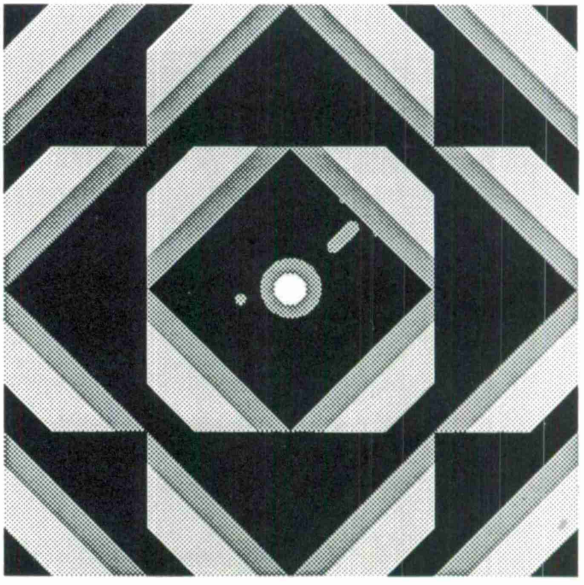
Most development approaches now use the incremental or build-release approach, especially in testing. From the baseline and design documentation, the incremental release functionality or build functionality is defined. There are a specified number of units or modules that make up the particular functionality, and as the program progresses we wish to monitor the way that the assigned functionality for a particular release might change over time. If it does change, it normally changes in such a way that many of the units are pushed over into later builds, and that is the thing we want to avoid.

Those are the metrics that ESD has been implementing. We are in the process of installing the metrics and collection methods on a number of programs. Metrics will be included in the RFP packages for all new programs. There will be either a Data Item Description or a paragraph in the Statement of Work describing information that must be supplied.

As for reporting the metrics, we're presently using two methods. One is to make it a deliverable, which does not really provide the interaction that we would like. The second method is for the contractor's program manager to present this information at Program Management Reviews. This gives some responsibility for that information to the developer's program manager, so that he can go to his engineering organizations and make sure that the information is correct. It also provides the opportunity for a dialogue between the developer's manager and the government program manager.

Session 2

Industry Views of ESD Software Acquisition



Moderator: Walter S. Attridge

Jack R. Distaso

Assistant General Manager, Systems Engineering and Development Division
TRW Defense Systems Group

I will survey the problems I have observed in our various programs, list some government actions and strategies that may follow, and then try to relate one to the other. I distributed a list of six software problems to about 15 current and past program managers, and asked them to rank the problems in order of significance. Every manager had the first three at the top and the last three at the bottom.

The first set of problems consists of the following: unattainable cost or schedule profiles; insufficient number of qualified personnel; and incomplete systems engineering/requirements definition and control. In many cases, either cost or schedule or both are already impossible when the bids go in.

Too many programs are in trouble the day the proposals are submitted, and sometimes when the Request for Proposal (RFP) is put out. The problems are sometimes directed in by fixed dates that must be met whether or not the requirements are there, or by unrealistic funding levels. As a result, the contractors become over-aggressive when they submit their proposals. A manager doesn't get promoted for submitting a realistic bid that loses. The result is that everybody immediately takes shortcuts, getting rid of tasks that may have to do with the methodology or the quality. After a while the problems begin to mount and eventually the people give up. Some other reasons could be just poor estimating or lack of understanding of the problem. Despite good understanding and good estimates, over-aggressive bidding can kill you right at the beginning.

The second problem is lack of the right kind of people. It often happens that because the contractor has a number of projects that all peak simultaneously, there are not sufficient qualified people to go around. Sometimes the problem is not having the right people at the right location, sometimes it's just poor judgment, and sometimes there is a shortage of people with particular skills. Right now, with the Strategic Defense Initiative (SDI) going on, all the sensor processing skills in the country are being stretched, and finding qualified people is getting tougher and tougher. If the personnel deficiency is in management, there is disaster right from the beginning; if it's in technical areas, it takes a little bit longer to discover.

The third major problem is not doing front-end engineering. Most programs are already in trouble at Preliminary Design Review (PDR). The basic quality isn't there because the requirements and design engineering activities are inadequate. You're not really dealing with user requirements because the user wasn't involved. Frequently, the government thinks that by not signing off on the requirements specification at PDR they're buying themselves some time to get their requirements better defined. This actually helps guarantee that the program is going to be in serious trouble. Sometimes the software seems to be in good shape at its PDR, but it is part of a larger system that is undergoing some significant technical changes. You must look at the larger system PDR to realize that the software requirements are changing, and that will impact the software downstream. An early unresolved problem will be a worse problem later. However, everybody tries to shortcut the system engineering because it is not as measurable.

You can sometimes get through with what looks like a good PDR, even if you haven't done your homework. The results are that you don't really have a model of performance; you don't really know what you're building; and the people aren't really geared up to do the exact job that must be done. What you end up with is a program in trouble and discouraged people who know they will be doing the same job three or four times over while the real requirements get defined. This usually shows up somewhere in integration and test, and the problem is often tied back to the lack of adequate system engineering before PDR.

The other three problems are similar. One of them is that the contractor doesn't have a disciplined methodology. The most common reasons are that the manager is either inexperienced, doesn't appreciate the value, or, even more common, knows better and just doesn't have the guts to tell his people they are going to have to go back and do it right. So they accept less than what is right. Often that doesn't get caught and that leads to trouble later. Obviously, if you have schedule and cost pressures, the first thing to go is methodology because the effects tend not to show up right away.

At times, you have a prime contractor and subcontractor who work differently. Although either style can be successful by itself, the clash between styles becomes a detriment to successful completion of the job. An example of this is when one contractor really wants a hard specification up front, while the other tends to use a prototyping or working group approach, defining things later in a controlled fashion. When you get the two together, the interface between them may not work and the project suffers unnecessarily.

The second problem in this set is unused or inadequate control, reporting, and review systems. There is a lot of talk about means of tracking progress, but if you have a poor system, you're going to have poor visibility, and you're going to get the wrong priorities on tasks. Again, inexperienced management may not appreciate the value of red teams, audits, and metrics. A more subtle problem is that you may have a lot of metrics, reviews, or other milestones, but they aren't real. If you don't have a process to assure that the quality is adequate at a given milestone, then it is probable that the milestone is only partially complete.

The third factor is the lack of adequate computer and environment resources. Most companies tend to use environments with which they are familiar, but if they must use an environment required by the system design that they don't have a history with, it can lead to some poor resource allocation. Sometimes the development environment is shorted to minimize the bid price, and often it is due to inadequate capitalization. Sometimes you size everything correctly, but because of the peaking process or schedule changes, you don't have enough resources. Sometimes the people don't know how to best use what is available. The end result of inadequate computer resources is lower productivity, discouraged people, and morale problems.

What can the government do about these problems? The first suggestion in Figure 1 is to use incremental and evolutionary developments. If you don't have good requirements up front, you can do some phased requirements definition and development to get user feedback to help you.

Several things can be done in acquisition alternatives; good, open communication is required so at least the bidders don't have an excuse for not understanding the problem. A competitive concept definition (CD) phase can make sense when the requirements are almost there and all you need is a little more time to design the architecture.

Integration contractors are sometimes a good approach if you're dealing with a system of systems; I don't think it's particularly good within a single system. Not everything has to be competitive. Sometimes, a contractor really does have a

Leverage of Government Actions

CHARACTERISTIC	INCREMENTAL/ EVOLUTIONARY DEVELOPMENT	ACQUISITION ALTERNATIVES	COST/ SCHEDULE OPTIONS	PROGRESS TRACKING	METHODOLOGIES/ STANDARDS	SOURCE-SELECTION CRITERIA	CONTRACTUAL OPTIONS
COST/SCHEDULE PROFILE	• BETTER DEFINED BEFORE IMPLEMENTATION	• SOLE SOURCE VS. COMPETITIVE • DRAFT RFP'S	• DESIGN TO C/S • C/S MODELING • SCHEDULE RESERVE	• SCHEDULING MODELS		• COST/ SCHEDULE REALISM	• FP VS. COST PLUS
NUMBER OF QUALIFIED PERSONNEL	• SPREADS TALENT REQUIREMENTS			• PERSONNEL PRIORITIES	• HELPS LESS EXPERIENCED PERSONNEL	• KEY PERSONNEL EVALUATION	• ELIMINATION OF FUNDING GAPS • GAINSHARING
SYSTEMS ENGINEERING/ REQUIREMENTS DEFINITION/ CONTROL	• WELL-DEFINED INCREMENTS	• COMPETITIVE CD'S • DRAFT RFP'S	• FSD DELAY		• ENFORCEMENT • 2167	• TECHNICAL UNDER-STANDING	• ECP PROCESSING
METHODOLOGY				• MILESTONE DEFINITIONS • TASK PRIORITIES	• PROTOTYPES • REUSABLE SOFTWARE • 2167 • TEST ENGINEERING	• REQUIRED METHOD-OLOGY	
CONTROL/ REPORTING REVIEW PROCESS	• TIMELY USER FEEDBACK		• C/S MODELING	• PROJECT PLANS • METRICS • INCHSTONES • RESIDENTS	• 2167 • ENFORCEMENT • TOOLS	• REQUIRED VISIBILITY	
DEVELOPMENT ENVIRONMENT				• RESOURCE PRIORITIES	• ADA • TOOLS	• REQUIRED ENVIRON-MENT	
OTHER BENEFITS	• USER INVOLVEMENT	• LOWER PRICE CONTRACTS			• TRANSPORT-ABILITY		• RISK SHARING

Figure 1

better approach, and perhaps working with him might get you a better program rather than always trying to play the competitive game.

The government could exercise some cost and schedule options. Sometimes it is better to hold off on full-scale development (FSD) until your requirements are better defined, so you have to face them up front. There are what I will call schedule reserves, which means changing the nominal milestones by adding months that can be handed out in reserve. If your reserve is given away by PDR, you know fairly early that you're in trouble, but at least it gives you some sort of management options. Better cost estimating guidelines and options would also help.

Design-to-cost/schedule acquisition approaches are useful, but are tough in a fixed-price contract. Many programs are lost during the bidding process because the low bidder usually wins. There should be some way of fixing the price and performing a technical competition to choose the winner.

There has been a lot of talk about progress tracking including the use of project plan definitions, milestone definitions, scheduling models, and metrics. In-plant residents can be very good if you use experienced people. They give good advice; they take a perspective that the contractor sometimes doesn't see because he's too close to the problem, and they'll work the problems back at the SPO. You can, however, cause more problems than you solve if you use inexperienced people.

The use of methodologies and standards is also important. Ada is going to create some environment problems and some environment solutions. One of the other speakers is going to be talking about DOD-STD-2167, so I'm not going to say much about it. Simulation and prototypes help you in getting your requirements defined in the early phases, even in the proposal stage.

Up-front test engineering brings some problems to light sooner. Very frequently we don't define how the program will be accepted until it is far downstream. By that time the mind is set, the people are in place, and the engineering that has been done doesn't support the real need. You could end up with a restart somewhere in the integration process. Both the contractors and the government are afraid to commit themselves too early to how the program is going to get accepted.

Of course, source selection criteria are important. They should include key personnel and methodology. We can't forget the technical design in our desire for better management and for the lowest bid; that can put the program in trouble before it happens.

It would be helpful if we could get rid of funding gaps. These gaps often occur when you have block builds, so it works somewhat against the evolutionary approach. One of the ways to lose key people is to have funding gaps, because good people will be snatched up by other programs or organizations. Similarly, long delays in getting the program started help create an environment where one team writes the proposal and somebody else executes the program. Approaches such as quicker processing of Engineering Change Proposals can sometimes get requirements defined and designed earlier.

"Gainsharing" is an approach where the government tries to encourage the contractor to share the wealth with the people, whether it's profit or some other kind of incentive approach. It can be a potential motivator.

You can match the problems and government actions and alternatives to see where various techniques would support or remedy some of the problems mentioned. For example, if you have a problem with the number of qualified personnel, an evolutionary development can help spread those talent requirements, and that helps overcome some of the peaking problem. If you have a good methodology or standard approach, that tends to help the less experienced personnel, and you can get better productivity out of them. In general, progress tracking and certain methodologies will give you some help across the board and if the government enforces progress tracking using metrics, you're going to improve control and reporting.

Elimination of funding gaps will also help you keep your key qualified people around. Using a better approach for cost schedule realism or using the design-to-cost, design-to-schedule kinds of approaches can help you overcome the overly aggressive bid or the difficult funding profile.

Sometimes these approaches have benefits for the government that are not necessarily directed toward these particular programs. For example, hard competition will generally tend to give you lower-priced contracts, sometimes to the detriment of the program. Methodologies and standards will often help transportability from one contract to another. Fixed-price versus cost contracting may provide more contractual options, but the more you go toward fixed-price, the more the government gets to share its risk with the contractors.

The following recommendations cover a number of these items and give some perspective on how I think the government might be able to encourage contractors and itself to do a better job.

- The government should acknowledge early that if they don't have good requirements and if they are not ready to sign on the bottom line, then an evolutionary development approach should be applied where at least an increment can be well defined and developed. The remainder can be developed after experience is gained with the initial increment.

- Unrealistic buy-ins, for whatever sets of reasons, are a primary cause of program failures. The government should consider using some sort of design-to-cost approach to help eliminate this problem.
- Add schedule management reserves that require joint contractor and Air Force concurrence to allocate. At the same time, provide incentives for meeting baseline schedules.
- Government should require more discipline and make milestones more meaningful. Make sure that the quality is there. Define milestones, so that they cannot be completed until quality is factored in and achieved. Enforce the methodologies and standards that you have agreed to use; make sure when you get through a PDR you really do have a system design.
- Help keep key personnel involved by eliminating funding gaps.
- The government ought to continue to push metrics and inchstones and force the contractor to increase visibility into the program.
- In-plant residents are very helpful when they are experienced; they can be detrimental when you get the wrong people on those jobs.
- Open communication before proposal submittal is very important. The more information all of the contractors have, the better they're going to do when they submit their original bid.
- Prototyping and simulation are very valuable tools to help you get a better handle on your requirements early on.
- Finally, government should get involved in early system engineering requirements development because that is when the problems start. Make sure that when you get through that process, the engineering is really done, including the test planning and test engineering. You then know how the program has been accepted, and I think you will have a higher probability of success.

Robert J. Kohler

President
ESC, Inc.

Most of my experience in developing software comes from my 18 years with the Central Intelligence Agency (CIA), so I'm going to talk to you mostly from my perspective during that time as a government software manager. In the last couple of years that I was in the CIA, we delivered nine software packages totaling about 6 million lines of code. All were delivered on time and within cost; seven worked to specification, one worked okay, and one was a disaster.

Of the projects that were successful, the biggest was about 2.5 million lines of code, and the typical project was about 750,000 lines of code. The largest project had 1,600 milestones over the four-year development cycle, and the typical project had about 640 milestones. The key is to lay out a plan at the beginning that is really good and manage that plan well.

Another key issue is how many government staff are needed to manage the job. We found one good person was perfectly adequate to manage about three-quarters of a million lines of code. The problem is finding the right person.

We are able to define requirements for hardware acquisition when we start, so I don't know why it's always so hard to do it for software. We solved the problem by spending a lot of time in pre-acquisition activities, and the performance against every requirement had to be demonstrated at Preliminary Design Review (PDR).

The things that went wrong with the disaster are not surprising. There was an inexperienced contractor. We put all the files in a commercial data base management system, and it shouldn't have been done that way. We sacrificed testing for schedule. The government's decision to save money resulted in the development on a machine different from the one the software was going to run on.

I think that we as contractors are trying to respond effectively to the government's need to have software development done within a reasonable cost, schedule, and quality, and meet the software acquisition requirements. Almost every senior manager in industry is terrified of software, however, particularly in the aerospace business. Not one of our corporate presidents has ever developed a line of code in his life, and they don't want to hear about it. The government's typical reaction that more management visibility is needed on projects in trouble elevates the project, and reports to the president of the company who didn't want to hear about it in the first place. That doesn't make a lot of sense, and doesn't solve the fundamental management problems.

Most government organizations don't know how to procure software, and don't know how to manage it, so the reaction on the part of the bureaucracy is to wrap more bureaucracy around it. Rules are written to help with procurement, and if things get worse, more rules are written. It's a very insidious process.

We have been at this business now for 20 years, and have been involved in the same discussion for 20 years. What we really need are some fundamentally new methodologies. Today's software technologies and methodologies do not fundamentally help the problem.

A few myths about software development are that everything would be okay if only I had a

good software manager, better people on the job, better ways of estimating lines of code, a better acquisition strategy, and more quantitative requirements. There is an element of truth in this, but since it is impossible to get good software people on *every* government job, the problem is essentially unsolvable unless you do something different.

My view is that there is a short-term solution and a long-term solution. In the short term there are two parts: pre-acquisition and acquisition. Too often in software development, the contract is let and nobody really knows what they have signed up for. This is absolutely wrong. When the contract is let, everybody — the government, the contractor, and the operators who have to use the software when it's delivered — must understand what their program is in terms of deliverables, in terms of schedule, and in terms of what it's going to cost.

The use of well-calibrated size and costing tools is also important. In the CIA, we used Price S, an RCA model. It worked very well, but it took two years to get it well-calibrated. People tend not to want to spend that much time getting the costing tool well-calibrated, but it's essential to do so.

Prototypes, algorithms, and methods do help in the requirements process. Too often the government tries to write all the requirements in isolation and when they hand the RFP to the contractor it's just too late. There must be a real dialogue between the people accomplishing this job and the using customer to be sure that the requirements meet the need, as part of the whole process before you go on contract.

Avoid specifications that drive implementation by telling the developer how to build the system. These kinds of specifications comprise one of the greatest faults of government program offices.

You must obtain the right resources: technology, funding, time, and people. When we negotiated software contracts, we added five percent to the cost and five percent to the schedule for margin. The objective is to motivate people to succeed, and you will not do it if you end up with a schedule that nobody believes they can meet and at costs nobody believes they can attain.

There are also technology margins. When you sign up for very sophisticated software programs, you must figure out if they're going to work, and what you are going to do if they don't, and build that into the program at the beginning. Technology is money, time, and people. If you understand where you may get into trouble, you can establish milestones for deciding if you are in trouble and, what you're going to do to get out of it. You need to establish fall-back positions at the beginning.

At the acquisition phase, never undertake a program that you know cannot be completed. Despite people dictating schedules from the top, you can say no. There's less impact on your career if you write a credible proposal that loses than if you take a contract that turns into a disaster. Have a well thought-out plan and stick to it, monitor the plan religiously, and react to problems instantaneously.

Establish decision dates for when you're going to invoke fall-back positions. Invariably, that is an important thing to do. Performance parameters must be demonstrated by PDR. And now for the government: don't over-manage, over-review, or dictate implementation.

Lastly, don't let a software specialist be the government program manager. He will try to make the software better, and as the industry program manager, you will completely lose control of the program. We made good system engineering people software managers. This worked well because they didn't know enough to be dangerous, but they knew enough about systems to manage the program well.

The long-term solution and the real solution, however, is technology. Ada and knowledge-based systems are two important kinds of technology.

At Lockheed, we found that our problems were inexperienced people, all kinds of different software environments, and vague requirements. We are building a system on our own called Plexus, which is a common software development environment.

Plexus will attack the documentation cost, because it will allow software to be developed without one piece of paper. When you finish the coding, the system automatically generates the specifications for you and software coders don't have to do that distasteful job of sitting down and typing it up. Trivial work, but when two-thirds of the cost involves trivial work, it should be dealt with.

Plexus is a near-term effort; it should be on line this year. We think it's going to give us a 25 to 45 percent increase in productivity, which really means reduced cost. It isn't attached to a mainframe; it's a series of netted PCs with a big data base controller that can be hooked up to the mainframes at the right time. It is intended to encourage sound software practices throughout the company and provide tools for the developers right at their desks.

We are also developing an environment called Advent, a joint effort between ourselves and Rational Systems, Inc., which has the only Ada-unique software development hardware that I'm aware of. We are developing an environment to produce very large and complex Ada programs using the Rational computer. The time it takes to develop software on that machine is much less than on standard machines. It's not a computer itself; it is intended to target software to other machines, though it does allow you to do development. The prompting and the automatic error detection are so good that people are actually forgetting how to code in Ada. They just sit at the machine and write software.

We are also working on a knowledge-based software development environment called Express, which will use domain-specific dialects. It will allow electrical engineers to code in electrical engineering language and mechanical engineers to code in mechanical engineering language. They have already developed the language, called Refine, which is in test. The goal is to allow rapid prototyping of about three million lines of code in about three months. Now 80 percent of that will be okay and 20 percent will not be usable, but that is probably a head start on what normally happens.

We should be able to have executable code written very quickly in this almost-English language and then allow the software to be maintained essentially at the specification level. In its final form, what you will do is continue to modify the code as you go; the prototype eventually becomes the deliverable version of the code. We think we're going to have this ability in about three years.

R. Blake Ireland

Manager, Software Systems Laboratory
Raytheon Company, Equipment Division

I suspect that most of us responsible for managing large software efforts and organizations are nearly overwhelmed by the rate at which demands are being placed on software engineering. I suspect that we as software practitioners have seldom allowed ourselves the time to stand back and assess what can and should be done to improve the system. Software management issues such as recruiting, training, and retention of quality personnel have not diminished at all with time, but I don't find that depressing because it is a measure of the vitality of our profession.

The inadequate number of software engineers with the necessary experience and skill levels is not only a general industry problem for today, it is also the leading cause of problems in software development. Therefore, the government's evaluation of a proposed contractor's software capability and capacity must be a key issue in contractor selection. If the contractor does not have the experienced software engineering talent available to assign to the contract, no amount of management legerdemain will rectify that deficiency.

Central to software acquisition is software management, which is finally beginning to be codified and understood. At Raytheon, we have tried hard to analyze each program as to why things went well and why things did not go well, so that the benefits of lessons learned will become our standard practice.

Let me briefly explain the software engineering function within Raytheon's Equipment Division. The laboratories are organized according to technology. The division is organized into business areas, called directorates, and functional development organizations, called laboratories. This allows us flexibility in grooming our engineering talent without regard to the rise and fall of contracts. The division's software engineering resources reside within my laboratory. Directorates have the responsibility for bidding and winning the new programs, for managing these programs, and for providing the program's system engineering component over the life of the contract. They assign development engineering tasks to laboratories and contract with the factory for the manufacturing effort once the product has been engineered. My laboratory actively participates in system engineering tasks, but in a support role to the individual directorates. We are in turn delegated full design, implementation, and test responsibility during the software development phase of the program. We discharge that responsibility by doing the work ourselves and by subcontracting to a very limited degree.

The Equipment Division has been responsible for the development of a substantial amount of software for ESD over the past dozen years. During the mid-1970s, studies were being done by IBM under Rome Air Development Center (RADC) sponsorship that led to the codification of what we now recognize as structured programming. The development of the first PAVE PAWS system in 1976 offered the opportunity for the first serious application of this methodology. A program support library was introduced, as were modularity and structured constructs. ESD in turn mandated throughput, memory, and storage reserves. Not unexpectedly, some difficulties were encountered that translated into mild schedule problems, but the overall result was encouraging. Meaning-

ful discipline was imposed upon the development process, and management, for the first time, had functional visibility into the development of the software.

Most of our current methodology traces its roots to this PAVE PAWS program. With only minor adjustments to the methodology, the program that followed, Cobra Judy, met or exceeded all performance goals and was brought in on schedule and within budget.

In addition to involvement in many ongoing ESD programs, my organization plays a significant role in all division programs having data processing content. These range from two-man technology programs to the development of software-in-the-large on weather radar data processing and air traffic control projects, each of which commands software development teams well in excess of 100 people.

We have accumulated quite a bit of experience working with major software subcontractors over the years. In managing these subcontracts, collocation properly belongs at the top of Raytheon's list. Software engineering does not lend itself readily to physical separation. The coupling of the software engineering function with the system engineering function is of necessity close and continuous since in most instances software binds the system together.

In addition, proximity to the hardware effort in tightly coupled applications is frequently mandatory. In those programs where we have subcontracted major portions of the software effort, we know that we will have a smoother development if the subcontractor moves his people into our facility, where they will be physically integrated into the program. We are, therefore, biased toward those subcontractors who are willing to collocate. Nothing can substitute for the management insight and control that come from having the software subcontractor under your roof using a common set of development facilities and tools.

Raytheon has also found it advantageous to lend software engineering personnel to on-site subcontractors to fill a small subset of the positions that the subcontractor would normally fill. Not only does this reduce the competition for local resources, but it also affords us an additional window into the development effort and built-in insurance should the subcontractor encounter difficulty.

To be effective, a subcontractor must be a working team member during the early definition phases of a program. Participation in the allocation of requirements for that part of the system for which he will be responsible imparts a level of system awareness to the subcontractor that extends well beyond the B-level specifications. Competing the software after completion of the requirements definition phase defeats this objective.

Finally, keeping a subcontractor properly focused on data processor reserves can be a real challenge. Raytheon has attached performance incentives, both positive and negative, to both memory and throughput. In every instance the subcontractor was able to earn maximum incentive fee and we were only too glad to pay.

Several precepts are essential to our successful software development efforts. We are firmly committed to establishing and agreeing to a complete performance baseline prior to entering the design phase. Software engineering participates fully in this baselining process. Full government participation in baselining is also essential, particularly when it comes to the locking down of user requirements.

We have found the incremental software "build" approach to be a key to developing successful systems. Our approach is a simple yet natural one that follows a "top-down" structure. We accumulate functionality into increasingly complex packages called "builds" and qualify each new increment, at least in a preliminary fashion. This qualified software is then available to rendezvous with hardware strings that are

being assembled in a similar fashion, thereby creating system "builds."

For example, in a typical radar program, the initial build would simply establish the run-time environment and perhaps provide some basic display control capability. This would be followed by a build that would close the radar loop, thus affording us a basic radar management functionality. Build three would provide sufficient tracking functionality to permit us to close the track loop and so on. Not only does the software "build," the system also "builds."

Significant benefits derive from this. Software flows through the system and through the development process in manageable chunks, and reaches a stage of architectural validation early in the development cycle. Of equal importance is that Raytheon and the government get early insight into and confidence in the integrity of the system. Of course, key to this approach are rigorous configuration management and traceability at both the software development and system levels that permit incremental qualification of each software build.

We have watched with considerable interest the work MITRE is doing in software reporting metrics to provide the basis for increased management visibility, and have incorporated a number of these metrics into our division reporting structure. Most levels of our management are finding development and testing progress indicators to be particularly useful in obtaining quick insight into project status. However, it is important that the metric being tracked be consistent from program to program and unambiguous. For example, the metric that marks the completion of detail design has a specific definition. This means that a design walk-through has been held, all action items against that design walk-through have been closed out, and the program design language under review has been transmitted to the appropriate level of the program support library.

Raytheon firmly believes that the government is a working partner. Our development methodology is keyed to the concept of disclosure and upward traceability at each stage of development, hence the role of the review and walk-through becomes paramount. Working-level participation by ESD and MITRE has become routine in these activities. We value their contribution to the review process, and we believe it affords the government true insight into the overall state of software. Similar benefits include working participation during the functional testing process.

We also endorse the ESD/MITRE concept of the software red team, but they should not be limited to programs in difficulty. Red team audits performed routinely at strategic points in the development process ought to have meaningful preventive value. Our internal red team efforts take the following form: The software management team presents us with a structured view of the entire software development strategy, such as organization, detailed plans, current status, methodology employed, staffing profile, baselines established, and tools in use. This presentation is highly interactive, and is followed by interviews with individuals selected randomly from within the development organization. These interviews help validate the effectiveness of the development strategy and often provide the red team with special insight. It is Raytheon's present view that these internal software red teams are more effective when carried out without government participation, with the proviso that findings and recommendations are shared in their entirety with the government; likewise, the results of the government red team audit should be shared with the contractor.

The procurement process, unfortunately, has built-in interrupts that have the effect of placing the integrity of the entire software engineering team in jeopardy. Typicaliy, we establish a cadre of software systems specialists during a major proposal or a definition phase contract. These experts play a key role in allocating the require-

ments, defining the interfaces, and setting the data processing architecture. If there is a long delay in the award process, we cannot afford to keep the software team intact. Thus, we sometimes find ourselves starting a contract with a team that has lost key members to other programs. I suspect that the overall system suffers greater damage when software teams are disturbed than when equivalent personnel relocations occur in other disciplines.

I have no packaged solution to this problem, but if a solution is to be found, it would appear to lie in the area of providing a modestly funded bridging vehicle that would allow us to continue work on the allocated software baseline while awaiting an award decision.

From a contractor's point of view, we sense a lack of government consistency in applying software standards and practices, not just service-to-service or even command-to-command, but also within commands. Unfortunately, ESD is no exception. Though DOD-STD-2167 should help, it won't begin to attack some of the underlying problems, such as the level of detail appropriate to B5 specifications and variations in test philosophies.

We commend a unique, innovative feature that ESD introduced: a periodic award fee that rewards superior, not just average, contract performance. An evaluation of performance against clear goals takes place every six months. Software schedule performance is a major evaluation category. It's not easy to win the incentive fee, but everyone from the software team to top management is aware of these incentives and is trying to win them.

We think the following policies and procedures can contribute to the successful acquisition of software by ESD.

- Software performance baselines consisting of B5 specifications and interface definitions must be approved at PDR and should be the joint product of the contractor, ESD, and the command.
- Meaningful government-level participation in design and code walk-throughs and testing should be encouraged.
- Consistency concerning software practice within and among government organizations should be pursued.
- The contractor should have a metric-based management reporting mechanism in place accessible to the government.
- Incremental software builds and tests should be required as milestones in the system development schedule.
- The management of computer resources through incentives and reserves should be an integral part of any program of a sizable software content.
- Periodic award fee contract features should receive broader use, to encourage focused management attention and to provide a formal opportunity for constructive interchange on contract performance.
- Finally, red team software audits should be carried out periodically and routinely, with open communication of results between government and industry.

I have never been more encouraged than I am today with our ability to provide sensible management control to the software engineering process.

Leonard W. Beck

Group Vice President and General Manager, Software Engineering Division
Hughes Aircraft Company

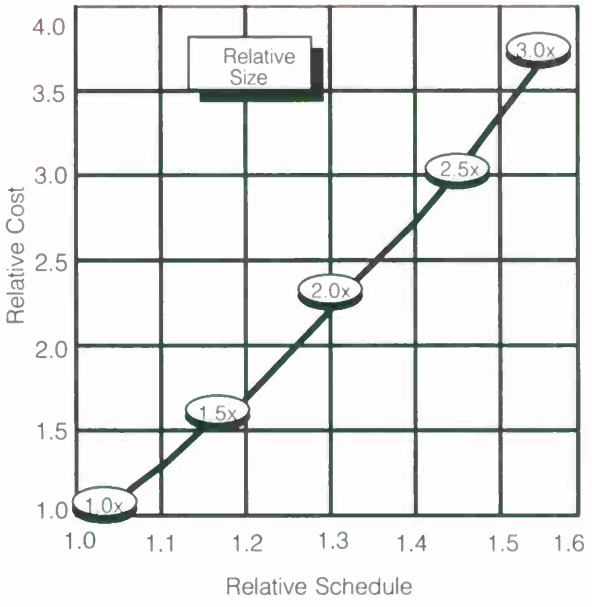
The basic problem in software acquisition is the difficulty of defining the requirements against which the software will be developed. The most frequent software risk and cost drivers concern requirements: complexity; high performance; and excessive, incorrect, and unstable requirements. The user, the contracting agency, and the developer each contribute to the difficulty in generating requirements. The inadequate requirements lead directly to cost and schedule problems.

The user, having to deal with real-world problems, asks for the best technology to solve his needs. However, the user is generally not aware of the cost and schedule consequences of that technology. Sometimes the user will ask for features that are desirable, but the cost far exceeds their value.

The contracting agency takes the high-level operational need from the user and generates the detailed requirements from which developers bid. The agency personnel have difficulty articulating precise requirements and may overstate them out of conservatism.

When the developer evaluates the requirements, he may see technical problems or conflicts with the specifications, or he may see modifications which would be cost-effective, but because of the competitive environment he is discouraged from making modifications that deviate from the specifications. The end result is that a set of specifications may contain overstated requirements, understated cost and schedules, incomplete or evolving requirements, have unresolved conflicts between them, and sometimes have areas that are not well understood, such as software reliability.

If Requirements Are Incomplete, Cost/Schedule Cannot Be Certain



Since the requirements are incomplete and uncertain, how can we as developers bid accurately on cost and schedule? The competitive environment does not encourage realistic estimates on cost and schedule. Further, if you had a baseline software size and you asked how doubling the size would impact cost and schedule, several models show that it would lead to 2.3 times the cost and 1.3 times the time for developing that baseline software. This suggests that in order to predict cost and schedule accurately we need good requirements.

Once the Requirements Are Completed, We Can Perform

	Programs				
	1	2	3	4	5
Year Started	1982	1982	1983	1983	1982
Original Schedule (In Months)	38	41	19	25	26
Schedule Performance Ratio*	1.10	1.15	1.21	1.19	1.54
Schedule Performance Ratio After SRS**	1.04	1.00	.88	.90	.80
Months Specs Late	3	6	6	7	10

*Ratio = $\frac{\text{Actual Schedule}}{\text{Original Schedule}}$

**SRS = Software Requirements Specification

Once the requirements are complete, we know that we can perform effectively. Hughes Aircraft Company has analyzed several comparable large scale programs that deal with embedded real-time systems ranging from several hundred thousand to over a million lines of code. The performed schedule ranges from 10 percent to 54 percent more than the original schedule. However, the ratio of actual to estimated schedule after software requirements specifications have been approved is much more accurate. This suggests that if we can assemble the right team early on in the program and have them work on requirements, we can improve our collective performance.

We are asked frequently to bid on a fixed price basis. At the beginning of the development cycle, there is a wide variance in the size of the programs that we must consider. As you go through the development cycle and are eventually ready to deliver, you know exactly what the requirements are. However, we are usually asked to bid the cost on fixed price contracts at an early point when considerable uncertainty still exists. What we need to do is find a way of bidding when those requirements are better defined. We need to develop some different acquisition strategies that will help us to collectively develop better requirements earlier, and that will enable a developer to bid accurately on a fixed price basis, on good requirements.

There are four alternative acquisition strategies: a Planned Evolutionary Development, a Customer/Contractor Team approach, a Cost Plus/Fixed Price combination, and Midcourse Reset.

Planned Evolutionary Development means that you develop only the well defined capabilities first. Defer the ones that you are not certain of, field an early capability, get user involvement, develop the next set, and repeat this process as many times as is appropriate. The benefit is that you will use your evolving system to get user interaction and feedback. The drawback is an apparent schedule extension. Many of our programs today use evolutionary development, but they are unplanned evolutionary developments. I'm suggesting that we plan for an evolutionary approach for the entire program. If you do that, you will improve scheduling over your current practices.

Another method is a Customer/Contractor Team approach. Here, the first thing under contract is to assemble a team consisting of users, contract staff, and the developer, and charge them with developing a good set of requirements. During this time, the team would use various methods to help define these requirements including models, rapid prototyping, or other appropriate tools. The team would be kept together until the software system requirements are approved. The obvious benefit is a direct dialogue between the parties concerned; the drawback is that somebody is going to have to be on temporary duty, probably at the contractor's facility. Also, you must have the right people to make this work.

The third approach is a combination of Cost Plus and Fixed Price for the contract. The less

well-understood and higher-risk tasks can be done on a cost plus basis while the remainder of the software can be developed fixed price. This has actually been done on some programs. An alternative version is to have a cost plus contract until the software critical design review, and then transition to a fixed price contract when things are better defined. The benefit is that it will allow the contractor to better satisfy evolving customer requirements; the drawback is that it is not clear how to estimate the fixed price portion on this first approach.

The last approach is basically resetting the cost and schedule for the rest of the contract at a selected critical milestone. This could apply to either a fixed price or cost plus contract. The benefit is that it will allow realistic cost and schedule management. The difficulty from the government's side is preventing the contractor from buying in if you know there is going to be a reset. I would suggest that at the beginning of the contract you might negotiate productivity rates, e.g., so many lines of code per person month, so much cost per person month. You could also say that the schedule will be determined by a model dependent on the size of the resulting software.

In summary, all software cost and estimating techniques are driven primarily by the size of the software, and size is the direct function of requirements. I presented four different software acquisition strategies. If used either individually or in combination, I believe they will enable us to achieve realistic software cost and schedules.

Ernest C. Bauder

Manager of Air Force Systems Engineering
GTE Government Systems

I volunteered to be on the DOD-STD-2167 Defense System Software Development Review Panel because I thought it would be a good way to invest a portion of my life. At the time, I didn't realize that it was going to be such a large portion of my life, but it has certainly been worthwhile. It has given me an insight into what many companies and government agencies are doing with respect to software acquisition.

I will spend most of my time talking about DOD-STD-2167, and specifically tailoring, because I think that is really the key issue. DOD-STD-2167 is one of the items that makes up the 2167 Software Development Standard (SDS) package, which consists of several components:

Joint Logistic Commanders' Joint Regulation — Management of Computer Resources in Defense Systems

DOD-STD-2167 — Defense System Software Development

DOD-STD-483A — Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs

DOD-STD-490A — Specification Practices

DOD-STD-1521B — Technical Reviews and Audits for Systems, Equipment, and Computer Programs

24 Data Item Descriptions (DIDs)

The JLC Joint Regulation on the Management of Computer Resources in Defense Systems consists of rules by which the government is to conduct its own business; it is being implemented by each of the services as they see fit. The Air Force folded it into Air Force Regulation 800-14, and that process has been underway now for about a year.

Then there is DOD-STD-2167 itself, a document of about 100 pages that received close to 10,000 comments. A fair portion of those comments, however, were on the Data Item Descriptions (DIDs). There is a complement of 24 DIDs, which are part of 2167, and also the updates of DOD-STD-483A, DOD-STD-490A, and DOD-STD-1521B. These revisions have only to do with software. We removed some appendices from documents 483A and 490A and put them into DIDs. SDS is a collective package that goes back about six years, and it is estimated that $10 million of industry and government work have been contributed to it. All of the industry work was done on a voluntary basis.

I think we succeeded because we were all equally unhappy. There is a balance between industry and government views. Government people feel that DOD-STD-2167 is terrible because it doesn't effectively give them the controls they need, and industry feels that DOD-STD-2167 is too restrictive. I think that DOD-STD-2167 represents a pretty good compromise. It is a single set of integrated tri-service standards.

There is a set of 24 uniform, tailorable DIDs, although many collapse into a parent DID or document and merge into others. Some of the management documents merge into the Software Development Plan (SDP). Most people feel that it's better than the existing standards. It also allows industry to develop tools and use them for a number of different agencies or tri-service groups.

Revision A of the SDS is continuing, and should be ready at the end of 1987. I am participating in the development of that document. A number of SDS issues are still outstanding, and they are the focus of a lot of criticism. System engineering, new methodology, Ada capability, and firmware can be considered a set. As we reviewed this document, we recognized the isolation of software from systems engineering. DOD-STD-2167 was developed by software people, and it does not really reflect the development of a total system. An appendix addresses this, and it is covered more thoroughly in the Joint Regulation. However, a lot of work still needs to be done, and we look forward to the update of MIL-STD-499 and other more comprehensive documents that would cover the total system development activity.

There are no new methodologies that are proven at this point. Therefore, rather than locking in on somebody's theory, we, as a collective government and industry group developing the standard, worked to provide alternatives to be invoked under the contract so it leaves room for technology insertion. Ada compatibility is a key, and the Ada community doesn't recognize we clearly wrote into DOD-STD-2167 that Ada is beyond DOD-STD-2167, and that it is left to the agency and the contractor to develop alternatives. Likewise, firmware is anything but firm, and, therefore, also employs the escape clause or the alternative approach.

Several issues cause substantial problems to industry: informal testing, excessive data, and software development files and folders. We have done our best in the time allowed to separate informal testing from formal testing so it can be handled in the tailoring phase. We think we have reduced excessive data requirements, and we have grouped Software Development Files (SDFs) into electronic files. If you have a 700,000-instruction system and you run about 35 instructions per unit, that is 20,000 units to handle in terms of informal testing, documentation of those units, development of SDFs, and tracking. That becomes a monumental task in itself, so we have included in DOD-STD-2167 the capability to handle them in other electronic modes. If that same data exists elsewhere in the electronic media, it doesn't have to be reflected in an individual physical folder.

There has been some criticism that DOD-STD-2167 inhibits automation. The stage for DOD-STD-2167 was set in the late 1970s, so that's really the technology baseline that the standard is written against. Also, DOD-STD-2167 does not cover Prime Item Specifications, classically known as B1 specifications.

Our biggest concern is blind application versus tailoring. General Skantze and General Chubb have sent out letters clearly directing that DOD-STD-2167 and its associated documents be applied, and both letters addressed tailoring in a number of places. It's extremely important to recognize that SDS is a full set or super set that has to be tailored down depending upon the particular application, and by default the government buys it all. If you do nothing and invoke DOD-STD-2167, you get every DID. It's intended to be tailored down.

During the review process, we argued from an industry point of view that it ought to be a minimum set to be tailored upward. The government argued rightly that the default set would then be inadequate. The burden is therefore on the people doing the tailoring to make sure that from the viewpoints of industry, the contract, the buyer, and logistic command, what is bought is what is needed so that the end product life is best served.

People seeking tailoring guidance can find answers in a couple of places. Draft Handbook 287 and Appendix D of the standard itself address tailoring. The significant feature of these resources is that they require the people doing the tailoring to have knowledge of the system application need and of affordability. It takes a very strong understanding of software develop-

ment and an experience base to buy only what is needed. That is true for both government and industry. There are ample opportunities in the acquisition cycle to get industry input, and that should be sought significantly by the government procuring agency. In the handbook and in the appendix there is an algorithm that describes the process by which one tailors, and again it presupposes that the person doing it has the required knowledge.

The government has recently initiated a DOD-STD-2167 advice hot line, (703) 276-2838. If you call that number, you will reach one of the contractors that has been hired to provide this kind of guidance. So far, people calling in on the hot line and those that I have talked to separately are concerned about the tailoring and the software systems specifications.

I was curious why we received so few comments on the System/Segment Specification (SSS), and now I know why. Nobody read it. As people are being asked to apply the SSS to real jobs, they are calling the hot line, and are asking where in that standard you write what the system really does. If you look at the old standards 483/490, you will find about 10 words that say to put the description of the system in Section 3.1, but any system specification has a fairly big section that talks about the total system.

I want to address tailoring from two dimensions: tailoring *out* of SDS and specifying alternatives *in*. The Statement of Work (SOW) is the driving document on a contract and it is the appropriate vehicle to tailor to the paragraph number. In SDS a lot of effort is structured so it's tailorable to the three-digit or four-digit paragraph number. But one must recognize the activities, the products, and the reviews that take place and only select those paragraphs that apply to that project, depending on where it is in 800-14 four-phase acquisition.

The vehicle for tailoring is a Contract Data Requirements List (CDRL). The right DIDs are selected, then you look at the DID backup sheets and provide the tailoring through them to buy what you need. And, of course, the SSS carries many of the design considerations and constraints that can be tailored in that way from the SDS package.

Specifying alternatives in is done through the Software Development Plan (SDP), which is the heart of SDS, and is where the contractor defines what will be done. In fact, that is the key to the flexibility that industry wants, the government controls that the government wants, and technological insertion.

The SDP is to be prepared as part of the proposal. Nowhere in DOD-STD-2167 do you see those words; they are in the joint regulation. The government is supposed to include in the RFP the requirement that contractors develop an SDP as part of the proposal. It's also supposed to be in the Contract Data Requirements List, so it will be delivered as updated at each phase transition. It may include the software Configuration Management (CM) plan, Software Quality Evaluation Plan, and Software System Programming Manual. They may either be separate documents or may be included in the SDP itself. That is an option in the tailoring.

During the process of developing the SDP tailoring approach, we had a lot of trouble deciding what happens when SDP changes are made. We were concerned that, as you reach a phase transition, you update your SDP, you refine it, and the government takes several months to approve it. So in 2167 it says "subject to government disapproval." You get approval of the SDP in the first place; then, if you make a change and update it, the government has the option to disapprove it, which we thought was a good compromise in terms of allowing work to continue.

You have to address non-deliverable software and the technology insertion items: firmware, development methodologies, and alternatives. The SDP is where you define them and get them

approved and brought in by the government. It is also clearly stated that you are to use modern processes and techniques.

Defining critical elements relates to phasing and when to do what in the life cycle. You can determine what kind of Program Design Language (PDL) or top-level design (TLD) description devices and methods you will use. A lot of people complain about the "top-down" design. Well, this is an opportunity to come up with an alternative. All of these things are called out to be included in the SDP.

Among the difficult issues are the software development files (SDFs). Their contents are to be defined in the SDP and not to be duplicated elsewhere. The real world has to address the design and coding standard for Ada because the ones in Appendix C weren't designed for, nor do they serve, Ada. There is a clear disclaimer on the DID as well as in the body of DOD-STD-2167.

How do you integrate? The contractor defines the integration in the software development plan. SDS is supposed to encourage automation of documentation. The key is that it takes knowledgeable contractor staff to define what should go into the software development plan and back it up with experience, which gives your plan credibility during on-site review. Both the reviewers and the people who are being reviewed have to be knowledgeable to recognize the value of what is being stated and whether it's honest, forthright, and realistic.

The software development plan will emphasize the need for knowledgeable people to do the tailoring on both the government side, in terms of the statement of work and the DIDs, and on the contractor's side, in defining clearly in an SDP what needs to be done.

I will now move to Commercial-Off-the-Shelf (COTS) hardware and software. We are dedicated to using COTS as much as possible, and we have had some successes and some problems. It takes a concentrated effort by both the Air Force and the contractor to see that this happens smoothly.

One of the problems is defining the system functional performance requirements versus the COTS actuality. This is where prototyping is valuable. It is important to run the software to find out what it will really do under certain circumstances. It turns out that how you use COTS software is probably more important than what the vendor bills as its ability to do a certain job.

Modification and source control are very key elements. Like everyone else, we have been burned by buying something, using it, and getting a new copy later that doesn't work anymore. Sometimes we were capitalizing on a bug in the software and we weren't even aware of it. When that bug goes, so does the feature. Similarly, new software should be identified so that it can be distinguished from prior versions.

Small software companies tend not to understand configuration management as we have come to know it. We need to select a vendor as part of our development. We must be very careful; and that needs to be recognized at the time COTS is specified.

Life Cycle Maintenance is also a factor. Much of the COTS software isn't going to be maintained for a long time, and both the contractor and government need to recognize that early on. My recommendation is to try some of these COTS packages in the concept exploration and the demonstration/validation phases and see if they really will do the job.

One of my favorite cartoons has the caption: "We don't know what we're making yet. We just started." That ties back to the need to understand the real user's needs. There are three vehicles for doing this: the operational concept document and maintenance concept document, the upper-level or higher-level documented system architecture, and the Computer Resources Life Cycle Management Plan. Those documents are supposed to come with the RFP. They rarely do. They are very good vehicles, and I strongly rec-

ommend Air Force Regulation 800-14 be followed.

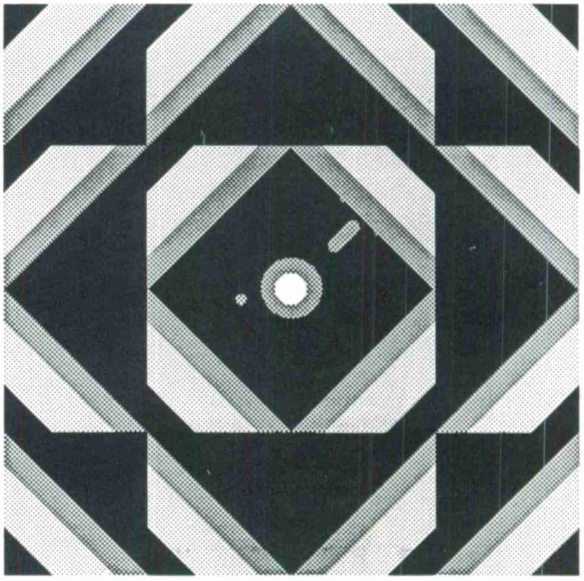
It's incredible how complex some of the systems are. That is primarily because the specifications grow by committee action. Contractors have to strive for simplicity and especially address the overall architecture. Use simple-minded terms and determine what these systems do and how the system interfaces with other systems. Some of these systems are huge. We don't recognize how many thousands of pages or tens of hundreds of thousands of pages of documentation have to be prepared, read, and maintained by the logistics command or user command if it has organic maintenance of that software.

The key point here is people. People have a limitation. You can only process a "mindsworth" of information. You either have to get people with bigger minds or break it down into smaller pieces and then structure it so the pieces interrelate simply and understandably. I found that if we can do that with the system in the architecture, it's amazing how everything falls into place — integration, testing, and so forth — through the development process. But if it's confusing up front, the result is going to be very confusing, expensive, and hard to maintain. So the key is to find the right people and retain that knowledge, skill, and capability.

Interestingly enough, there is more turnover on the government side than on the contractor's side. We still have people working on projects who started with them back in the mid-1960s. On some of the projects, in the last two years, we have a new set of faces on the government side.

Session 3

ESD/Industry Dialogue



Moderator: Alan J. Roberts

Alan J. Roberts

Senior Vice President and General Manager
The MITRE Corporation

As I was listening to the experienced people speaking here, I wondered what we have been doing. We have been working at this for 30 years, yet most of the programs are still in trouble.

In looking at all the problems we are having, I wondered what it would be like if we hadn't been here for those 30 years. Today, at least systems are being delivered; I worked on a number of programs 20 and 30 years ago that never got delivered. Most of the time now we are delivering. We are finding ways to deliver and understanding better why we don't deliver.

For a long time, we have been talking about techniques that are 10 to 20 years off in the future as though they were going to change the next program we had in mind. I think now that we have a better understanding that this is not so.

There were some points made in the opening sessions that I would like to emphasize. If we don't believe the evidence of why we are not delivering systems on time and within cost, then writing better specifications and doing better cost estimates will not happen. If we don't have information, we can't write a better specification. If we must base everything on unknowns, we are not going to get better cost estimates.

I think we must believe the evidence. We have to find ways to live with all the hard jobs and problems of producing software while we wait for higher order languages and knowledge-based or expert systems, which will change the process in a dramatic way.

The project officer or program manager has a difficult job. We have heard that every program starts out with too little money, an impossible schedule, and too little hardware; often we don't know what we're trying to build, yet we expect to deliver on a schedule and cost estimate that was pulled out of the air. Some of the contractors are able to find a way to work through that maze and get something accomplished.

In my experience, the way that is done is to find a way for the government and industry to get together. They are then able to find a way in which both of them will benefit. There isn't one successful program I know of where there was a strong adversarial relationship between the government and industry. We have to know what we are dealing with and take the proper action.

If we are all in agreement about the problems, why are we not getting on better than we are? We have said that we know what we ought to do and how it ought to be done, but we are not doing it. We need to get an agreement on how we will proceed and then go out and do it.

Session 3 Panel

Session 3 provided an open forum for public interchange among the speakers of Sessions 1 and 2 and the audience. The speakers, representing government and industry, had presented their views of the problems they had perceived in ESD software acquisition and the solutions they recommended. This session's discussion was an analysis of well-recognized and longstanding problems and a search for solutions, rather than a confrontation between two opposing viewpoints.

Requirements were singled out by speakers and audience participants as the most perplexing problem for C^3 systems. There was unanimous agreement that it is necessary to define requirements and control them in order to estimate the time and cost of software development, to design and implement software, and to be able to test and validate the software against the requirements. Yet most people felt that requirements are seldom well-defined at the start of full scale development. Proceeding with design when the requirements aren't firm was cited as the main reason why software is delivered late.

Defining software requirements more clearly could be achieved by doing some design and implementation work to see what's feasible, especially if the job is new. Rapid prototyping, simulation, and executable specifications might be useful means for firming up requirements. These tools show the cost and risk, and demonstrate functionality to users and developers. One method to determine whether requirements have been adequately defined is to specify how each one is going to be verified before baselining them.

Industry would like to see more user-developer communication on the specifications and more time to see that each party has the same understanding of the requirements. All people involved should sign off on the specification. This includes people who will implement the software, people who will test it, people who will maintain it, and people who will use it.

Another problem with defining requirements up front and controlling them is that the requirements change frequently. This must be anticipated — the process must allow for control and for change. A product that is delivered within budget and on schedule is a failure if it doesn't meet user performance requirements.

There is some data to show that if requirements are well defined, substantially less time is needed to complete the software. However, government program offices are wary of stretching the overall schedule, so the time and resources needed to more completely define requirements are frequently not there. If the government could speed up its contracting process, the schedule could be shortened and more time could be allowed up front.

Another problem, cultural in the Air Force, is that the usual length of an assignment is four years, which probably isn't long enough for a complex program. This often limits the continuity of management that is strongly recommended for industry development organizations.

There are other approaches that industry is looking into to improve productivity in software

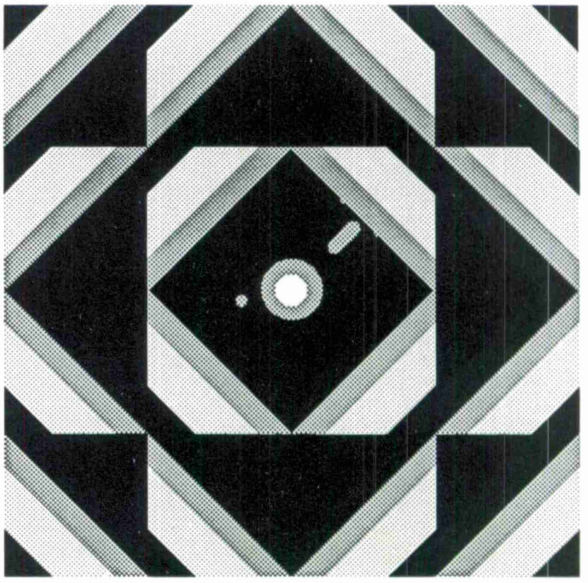
development. One of these approaches is to find ways to let a system engineer translate requirements into a system more directly without programmers. The Japanese are proving that reusability leads to 10 times our productivity. If something already does the job, use it. The military may have to start buying product lines instead of custom-built software.

No discussion of software problems is complete without some comparison to hardware. The software people had the last word in the session, but what they advocated was having the first word in the system design process.

It was noted that everybody believes you can fix all the hardware problems with software, so the hardware is tied down and then the software people attempt to make the system work. If it doesn't, it's a software problem. The mistake is in comparing development software, which comprises almost all of the software that ESD buys, to off-the-shelf hardware. When the day comes that we write the software first and give it to the systems engineers to wrap hardware around in order to achieve system capabilities, then we can talk about why the hardware and software development processes should be the same.

Session 4

Ada and Software Development Environments



Moderator: Christine M. Anderson

Charles W. McKay
Director of High Technologies Laboratory
University of Houston at Clear Lake

For the past three years, I have been privileged to work with an advanced research team composed of about 100 representatives from 30 industry organizations that traditionally support NASA. When we began three years ago, there had been more than a decade of studies involving personnel from all NASA centers concerning the space station. They had concluded that a fully functioning space station program would represent an integrated end-to-end information environment.

The challenge that my team was tasked with was to go beyond what we had learned in programs like Mercury, Apollo, and the space shuttle to look at those truly dark areas with regard to the space station. We had to identify those things that are either new, things that we do not know how to do, or things that we do not know how to do very well, and to illuminate them and to try to reduce the risk to the maximum extent possible.

We were asked to assess the possibility of applying the tools of the Minimum Ada Programming Support Environment (MAPSE) and assess whether they would be adequate to support the full life cycle of the space station software.

Let me give you the conclusions first. Is Ada ready for use in C^3 systems? In the opinion of my team, the answer is yes. Certainly it is ready for the design of such systems. Whether it is ready for the development of such systems depends on whether the system is a single embedded processor application, a multiprocessor application, or a distributed network, particularly one that involves real-time and data-driven applications.

For the design phase, which we think is far more important than subsequent phases in software development, we think Ada provides a very good base for operating in a software engineering environment. For the rest of the phases of the life cycle, we felt that the MAPSE does not support large, complex, non-stop distributed applications to the degree we would like. This is not a criticism of Ada. This is a criticism that applies to our industry, by and large, regardless of what particular language you may choose for software development.

The second question I was asked to consider for this symposium was whether ESD and industry are ready to use Ada. My own opinion is that we are not ready but it is time to proceed anyway. We need to make progress. There are many accusations being leveled at Ada that I feel are analogous to shooting the messenger that brings the bad news. Ada is revealing problems that have existed for years. We have not come to grips with them. Ada is putting some things on the table for people to view. Many of us are uncomfortable with the fact that we are not appropriately educated to do the things that we believe need to be done.

The context of this point of view is requirements like those of the space station. It is a program that is intended to evolve over 10 to 30 years. It is a large, complex, distributed computing application with a long schedule of modular growth. It will involve distributed hosts and a very large collection of distributed target computers. There will be many developers in the United

States and around the world. There will be an integration, verification, and validation host and test bed. In fact, our informal estimate is that this is probably the largest computing project solely for peaceful and scientific purposes that has ever been proposed. The challenge of our team was to look at computer systems and software engineering for such applications and to reduce the areas 'at risk.'

To give you some idea of the requirements for the space station program, it would have ground stations, free-flying platforms, and space stations, as three instances of 12 types of local area networks. A fully configured space station would have 23 clusters of computing requirements. It would have a 10- to 15-year development cycle and a virtually infinite life cycle. All of the subsystems would be partitioned to isolate implementation details of different components from one another. How services of other components are provided should be totally transparent to contractors who are developing a component. We looked at some of Ada's support for modularity, for information hiding, and for abstraction.

We asked what life cycle support should a programming support environment provide for large, complex, distributed computing applications such as space station systems. The question that we were given as a team three years ago we have answered to our satisfaction and hopefully to the satisfaction of NASA. Ada has been baselined for the space station program, not only for the components that support host development functions, but also for support of the target systems.

We were concerned with evolving an appropriate systems and software engineering environment to support this evolving program. The team observed that the environment must support more than software. We defined the life cycle model for a software support environment (Figure 1).

You must begin with systems engineering, which is followed by software engineering, which is followed by hardware engineering. These are intricately interdependent. Details of managing people, logistics, and project planning and control are involved. At the heart of the systems and software support environment is an information system that has a project object database that persists over the life cycle to capture the design details in the system, software, and hardware engineering. It also captures all of the details that led to design decisions and the trade-offs involved. Configuration management is appropriately integrated along with quality management.

There must also be provisions for the management of reusable components in the design phase, not just reusable source code. As a team, we believe that the reusability of source code may ultimately be far less important than the reusability of design and the other products of the design process. If you can accurately trace from the systems requirements through the software and hardware trade-offs, and how each succeeding phase evolves, then you may find that there is more value in terms of reuse than can be found in source code libraries. This possibility is a tribute to DOD-STD-2167. We are 15 to 25 years behind in the state-of-the-practice in software engineering compared to what we know how to do. A standard life cycle model will facilitate improving methodologies, tools, and such system-level attributes as traceability and reusability.

It is incumbent on us to adopt good standards and to tailor them for our benefit. This is a strongly typed approach to a software support environment. For instance, P1 in Figure 1 is the requirements phase. The requirements are intended to be captured in the project object base as strongly typed objects. They are operated on by a finite set of technical tools and management tools. The only access to the objects in the project base is through permission and authorization to use the tools that manipulate the objects.

It is important to remember that the space station will evolve over 30 years. The environment has to be nonstop. We don't have the luxury

A Life Cycle Model for a Software Support Environment

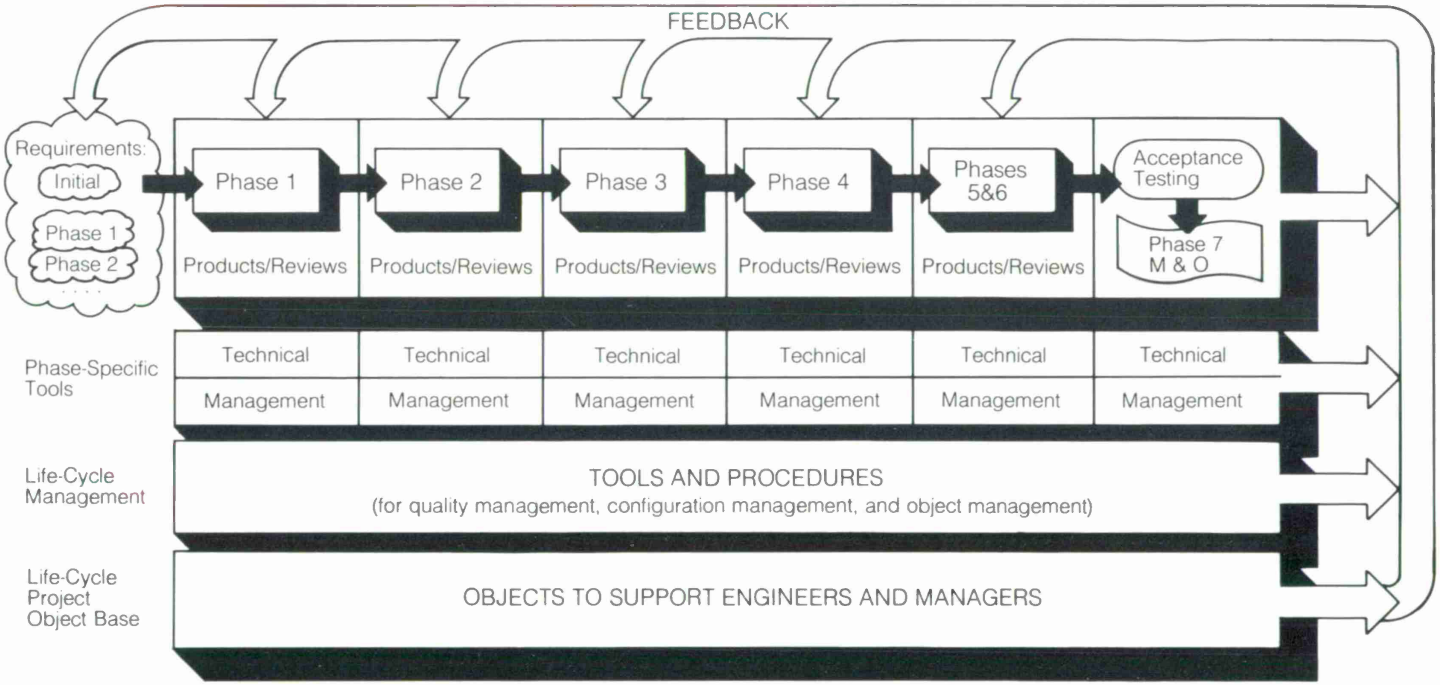


Figure 1

of the shuttle program to build software, simulate and test it, load it on the bird and fly it, and at the completion of the mission, do an autopsy. Imagine instead that there are unattended components, that changes in software functionality must be directed from the host environment, and that we will be expanding the system over time.

As you trace through the activities in this strongly typed life cycle model, you have to determine if the environment has the capabilities necessary for our systems life cycle. For example, consider Ada run-time environments. The first ones built were for a single processor executing a single program. It was unfortunate that almost none of the first run-time environments was capable of being extended to multiprogram support. If there had been a more powerful environment, then by setting the number of programs to one, we would have had everything in the first environments. As we move to multiprocessors, it is an indictment of our approach to implementing Ada compilers today that compilers tend to target a single processor. We are working with some compiler manufacturers to create run-time environments that will allow the distribution of Ada entities to exploit true parallelism.

We are also concerned about the issue of fault tolerance in distributed networks. We have no alternative but to think from the very beginning about this issue or we will have deadlock or live lock in the target systems.

This gives some idea of the kind of environment we envision. We expect host environments to be provided to various contractors and NASA centers throughout the world. All source code would be processed and receive final verification before deployment at an Integration Software Support Environment facility. Here it would be compiled, checked against the project object base to see what components of hardware and software already exist in the target environment and what the effect of the new software would be on the workload so a test plan could be generated. A check would be made to see that the functional and non-functional requirements are met.

In conclusion, we have the opportunity and need to extend the technical tools and the management tools of the Minimum Ada Programming Support Environment far beyond today's definition to be able to support a very large and evolving program from which we think this country will benefit greatly.

Gerald E. Pasek

Program Manager, MILSTAR Ground Segment
Lockheed Missiles and Space Company

The MILSTAR system is a DOD communication system. It consists of satellites, space ground terminals, and control terminals. I am directing the Mission Control segment, which consists of several hundred technical personnel who are working at Lockheed to provide satellite servicing and system status reporting. This system must be survivable and supportable by military personnel.

The challenge is one of applying in a design-to-budget program available hardware that we call "off-the-shelf," which means utilizing products from vendors that are being applied to multiple projects and limiting the amount of project-particular tailoring and design. By using off-the-shelf hardware, you can get the vendor to put in his own IR&D money to develop the hardware and thereby lessen the cost to the project. The more you tailor your hardware to the project, the more you are going to pay for the entire system.

The other challenge on MILSTAR is that the platforms may be aircraft, ships, or trucks. In the case of the airborne platform, every pound of weight added takes away a pound of person or a pound of fuel, since the airplanes usually fly at full gross weight. Therefore, we want to minimize the weight requirement. In addition, the platform is not dedicated to the MILSTAR program. Our role is to control the satellite in the system as part of an overall mission complement. This means that we are on various types of mission platforms where we are being carried only in the event that we are needed to support the MILSTAR system.

The environment is DEC-based, using next generation state-of-the-art 5¼-inch Winchester disks. All hardware is fully militarized. The disks can store 140 megabytes on what amounts to a cigar box size package. This allows us to easily transport the disks via mail or courier techniques. We are currently running on a full complement of the disks, which are packageable. A 19-inch by 15-inch disk package gives you 650 megabytes worth of data, which I think is more than adequate for most software databases.

The system has a single operator per platform. The NASA Space Station environment described by Dr. McKay includes hundreds of people involved in satellite support; for us to break a system down so that it is supportable by a single individual is, indeed, a challenge. Our goal is to develop a system that will allow one person, with reasonable training and retention, to support the system.

From a survivability standpoint, MILSTAR is a distributed system. We will have many platforms, any one of which can support the system. Fixed ground facilities provide for centralized planning, configuration management, and software development and maintenance. Without centralization of the software and the database, a distributed system would never work.

The system weighs about 1,000 pounds. The MILSTAR operator display is a plasma display that uses a joystick to control a cursor. We did not use a touch panel system because that system is somewhat difficult to operate in a violent environment such as a plane that is flying through turbulence or a ship that is bouncing around through the waves. In addition, the display is multi-windowed, similar to a Macintosh

type of operating system, so you can see what is happening in many different places.

The software that we are building runs on the MILVAX, so all our development is done on commercial VAX systems, using MIL SPEC 483 and 490 documentation. The operating system is VMS. Development is underway for all of those pieces of software that we need to run the system: real-time processing and control, orbit management, system management, data and display support, and diagnostics that tell the operator what is happening. There is also a set of fixed ground environments for software and database support, configuration management, planning and archive management, and system simulation. In the event of a system failure, the operator in the field is tasked to keep the system going. Once the system is running again, someone else must find out why that event occurred.

This software comprises several hundred thousand lines of code, written in Ada and FORTRAN. The reason for FORTRAN here is that we believe we may encounter areas where Ada will be a constraining factor, in which cases we have agreement with the government that we will insert FORTRAN.

Our development environment is in place. We are DEC-based, including Ada workstations. We have Rational Systems equipment in place. We have an in-house training program for our own people. We also have Ada experts who are available on an on-call basis to consult with people.

We have recently completed our Part 1 specifications. It generally takes a couple of years to finalize Part 1, particularly where you have, in my case, 30 different agencies involved in the program.

Part 2 specifications are being developed, and we are now trying to tailor the documentation. As you know, MIL SPECS 483 and 490 were not written in the Ada environment. We must therefore do certain kinds of tailoring of the documents to represent the Ada considerations in the documentation. Designs in an Ada Design Language (ADL) are also being completed. The objective is to come up with a detailed, compilable design. The tools are being developed to take that ADL and develop various types of documentation.

Our longer-term production development environment is VAX-based with multiple types of terminals attached. This kind of environment allows us to use various levels of compilation. With a MicroVAX system Ada compilation on the order of 300 lines per minute is possible. The larger VAX systems can be run at approximately 1,000 lines per minute, depending on the type of compilation. Rational Systems provides an Ada compilation engine that runs 2,000 lines a minute and can do partial compilations at a very high speed. Documentation on the various kinds of terminals is resident on disks. A copy of this documentation can be produced on the laser printer. This allows system engineering to access the requirements, do traceability, and go through the configuration control process.

We have a system simulator which represents to our Mission Control Element (MCE) what the external world looks like. Our militarized equipment would connect into one of these VAXs for data and satellite-type generation. We will use the simulator to test our environments and the system. This ensures that we are ready to support it before we interact with the satellite. This is important because I have seen many systems crash once they get out into the real environment.

Why did we go with Ada? The MILSTAR system is expected to be operable into the year 2010, if not beyond. We must therefore consider where the software environment is and where the software environment is going. In 1984, with the permission of and funding by the government, we conducted a study to look at long-term projects and assess our course of action for systems that we will be developing in detail in the mid- to late-1980s and maintaining through the 1990s and into the year 2000. The system that was chosen was Ada.

JOVIAL is not a viable alternative. There are no truly good compilers around that are being supported by the vendors, and we have had a lot of trouble with JOVIAL compilers. FORTRAN, on the other hand, lacks the true standards, and thereby restricts portability. My management asked if we could save money by going back to FORTRAN. The answer that I have been able to detect from the programmers is that the same kinds of problems exist with FORTRAN after a certain point.

One of our goals was to use state-of-the-art equipment when it became available. We presume that computers will increase in memory and become faster and cheaper as time goes on. We wanted a degree of software portability in that sense. Ada is also forcing a disciplined design structure, which will ease the integration and maintenance over time.

Everyone recognizes that the Ada environment has too few people out there who are considered experts today. The typical Ada person that we encounter is relatively junior in nature, recently out of school, and does not have a lot of hard design experience. The design experience that they do have was in the academic domain, where time and budgets were not a problem. We found that with a typical programmer, particularly one who is adept in Pascal, learning to program in Ada is accomplished quickly and smoothly. In one month a programmer can learn the language and be reasonably proficient in it. One of the problems is to find people who understand globally large systems and can work and deal in that environment.

In sophisticated environments, communications among people can be a problem. The programmers become so accustomed to their terminal that if the information does not come at them from the terminal, they do not know where to find it. I have a great deal of difficulty getting people to move 20 or 30 feet to talk to somebody who may know the answer. Ada does not solve the peer group communications problem. As with any software development effort, development under Ada requires very strong leadership, firm direction, and very close monitoring of activities.

I have some recommendations relative to Ada:

- Prototyping should be performed early because Ada code can become a problem from a time and memory perspective.
- Modeling, with attention to sequence, should be done on control and internal software communications and database access techniques.
- Test the prototype to assess specific Ada implementation overhead and throughput performance on selected or simulated selected hardware.
- Study the selected Ada compiler and performance of its output code under an Ada tasking environment and a local operating system environment without tasking.
- Provide software designers with study results and development guidelines tuned to the specific Ada being used and the system characteristics.
- Assure that lead programmers have "hands-on" experience with non-trivial programs developed in Ada with similarities to the present system. Alternatively, have them do the prototyping and tuning of the prototyping early.

Now to answer the questions. Is Ada ready for use in C^3 systems? I believe it is. Are development communities ready for Ada? Well, maybe. My senior management still does not understand the problem; they are in favor of taking the safer and proven approach. Are program managers ready? I am not sure program managers are ever ready for large software development projects. Therefore, it will remain a question that Ada certainly won't solve.

Nelson H. Weiderman
Senior Computer Scientist
Software Engineering Institute

I have been working on the Evaluation of Ada Environments project with a small group of people since last summer. The first thing that we set out to do was to define some criteria and a methodology for evaluating Ada environments. In a nutshell, the methodology is to define some tests and experiments or user scenarios that are independent of environments, and use the tools of those environments to get our results. We have finished the methodology definition, and we are now applying this methodology to several environments. I don't want to report today on any specific results because the project is ongoing, but I would like to tell you a little bit about the things that we have learned and the general state of environments as I perceive it today.

To begin, I would like to say a few words about the Software Engineering Institute (SEI) and put this work into context. The SEI is relatively young; we have been around for just over a year. We are comprised of about 100 people, and our major focus is technology transition — getting the technology out to the contractors that need it. We are doing work both in technology areas and in non-technology areas. We have a legal project, management projects, organizational projects, and behavioral projects.

Obviously, Ada is a very large part of what we do, and we treat it not only as a language but as a culture, including many software engineering techniques and principles. We are also concerned with infrastructures for open and integrated systems. We want to find ways of incorporating new tools into environments and making those tools work together. Dr. McKay spoke of persistent information that lasts for the lifetime of a project and has to be tracked along the way, that is, persistent object bases. Automated reuse will be important for design and evolution, as will artificial intelligence and expert systems.

We at the SEI begin by stimulating these technologies and doing some research, followed by acquiring and exploring the technology. We then refine and integrate, perhaps doing prototyping. Next, we will install products in our environment and use them in a production fashion at SEI, and we eventually disseminate products out into the real world.

The Ada language is obviously fairly mature. The maturing process took place over a period of about eight years, from 1975 to 1983, when it became a standard. This standard will be in place for four or five years before any changes are allowed.

The compiler situation dates back to the completion of the design around 1980, when people started working on compilers. The first Ada compiler was validated in 1983, and we currently have 29 validated Ada compilers. The compilers have reached a state of maturity, and the amount of change in the design is diminishing. I believe that we can say that we have production-quality compilers available today.

The situation with support environments is that we had a series of requirements back in 1979, culminating with the Stoneman requirement. In contrast to the language, the requirements for the environments were not nearly as well designed. There was not nearly as much input or response to those requirements, so we

really cannot compare the environments to the language requirements. We have made some progress, but there is a great deal more progress yet to be made.

There is some irony in the fact that although Ada was developed for embedded systems, it has received acceptance first in the information systems area, and only recently has it been employed in embedded systems. In fact, out of those 29 compilers I spoke of, only three are cross-compilers for embedded targets. There is one for the 8086, one for the 68000, and one for the Z8002. This is a relatively young area.

Distributed embedded systems are comprised of more than one host development environment and more than one target. We have seen very little activity in this area, although there are about six families of multiprocessors now with names like Sequent, Tolerant, Flexible, and Convex. These processors support distributed host development.

Ultimately, environments must support heterogeneous distributed non-stop embedded systems; heterogeneous, meaning that the processors in this distributed system have different instructions and architectures, and non-stop, meaning that we must have environments that support these run-time systems that change on the fly. Such environments will not be available for some time.

We have identified three different kinds of environments. A layered environment is one in which we build a user interface around an operating system that already exists. There is another kind of environment that I will call incremental, in which we take an operating system and add Ada tools, such as a compiler, a debugger, and an editor. The third kind of environment is an Ada machine, constructed by building an Ada environment on a bare machine, possibly using specialized hardware. The Rational machine is an example.

We have defined four broad categories of criteria for Ada environments. The first criterion is functionality, which has to do with what tools are in the environment and what features are incorporated into the tools. The tools should cover the entire life cycle from design and requirements definition to testing and analysis and project management. We must determine whether the environment supports all these activities of the life cycle.

The second criterion is the user interface. The most obvious component of the user interface is the generalized command language, which can be a menu-driven system or a graphics-driven system. Other components of the user interface include the on-line help system, the documentation that goes along with the environment, and options such as the ability to view different objects in various ways.

The third criterion is performance, something that we are all interested in, and this involves the efficient use of time and space resources, as well as response time to users.

The fourth criterion is the system interface. The term interface refers to how well this environment communicates with whatever is beneath it, be it another operating system or a bare machine. Our goal is to have the environment make use of all underlying capabilities and not hide any function unnecessarily.

The quality of an Ada environment depends upon all of these criteria and their interactions, as well as the freedom from error and the robustness, or the stability of the environment. Defining these criteria therefore becomes a process of iterative refinement.

We have learned in the short time that we have been in this business that checklists of functionality are not sufficient. The Stoneman requirement for environments and the definition of a Minimal Ada Programming Support Environment did little more than indicate which functions ought to be available. It did not detail which features should be in those functions and how

those functions should work. The requirements indicate only that the environment should have items such as an editor and a debugger, for example.

Some of the core functionality of the environment cannot be added easily. You must start with a basic infrastructure, which would include the database and some of the other tools for integrating new capabilities. You can't simply take a collection of tools, put them together, and call it an environment.

The functionality of an environment can greatly influence productivity. Obviously, you want an environment to do as much as it can. If you have an Ada unit that is recompiled and there are dependent units that have to be recompiled as a result, you don't want the environment only to tell that you have to recompile these other units. You want the whole process to be automated.

To date, the environments that we have been looking at have not demonstrated full functionality. There is much that is missing, particularly in the areas of design and project management. We feel that Ada environments have a long way to go.

In our experience with user interfaces of these environments, the command languages have not been particularly friendly. They are based mostly on 1970s technology for environments; they are still command-oriented. Generally, you need wizards to operate in these environments.

Ada has been proposed as a command language for environments and I am not convinced that is the correct answer. Ada was designed to be read, not written, and command languages should be written, and do not generally have to be read. If Ada is used as a command language, then the environment should provide a great deal of help with the syntax.

The environments must have systems for interactive use, such as command completion, wild carding, and abbreviations; items that we have come to expect in operating systems. We have found that both error messages and documentation vary a great deal in quality, and that many error messages are not helpful. They do not identify the source of the error, resulting in cascading error messages. These are things that we should not encounter in the 1980s.

With respect to performance, one of the indicators of maturity and stability of a technology is the ratio between the best and the worst environments, and we are still seeing very high ratios between the best and the worst, up to an order of magnitude. I would expect that as the technology matures, that figure would come down to the 30 to 50 percent range.

The performance of a system greatly affects the style of development. If you have a simple operation such as creating a library that takes 30 minutes, people are not going to create many libraries. This is a problem in some environments.

I am very skeptical about lines of code (LOC) measurements because there are so many different ways of measuring lines of code. Most vendors are kind enough to eliminate comments and eliminate blank lines when they talk about lines of code speeds of Ada compilers, but some vendors still use the number of lines on which Ada appears as opposed to the number of Ada statements. In a large number of sample programs, this ratio is about two lines on which Ada appears to one Ada statement. You must also take into consideration what is being compiled; that will have a great influence on the speed of a compilation.

Incremental compilation and automation of compilation are very important issues, and can be more significant than raw compilation speed. If you make a very small change to a large program, and the environment can do that easily, rather than compiling the entire unit, then you are much better off. We are starting to see this in some of the environments that are now coming out.

A slow interactive system is equivalent to a batch system. If you take more than 30 seconds to do a particular task, the developer is going to get tired of waiting, and will put that system aside.

In terms of system interface, we found that the layered approach can be costly in terms of performance. Another problem with the layered approach is that useful functions may be hidden under this layer that you have created and they cannot be accessed or are difficult to access.

In addition, I have seen no evidence yet that the layered approach has met its objective of providing portability. Integration is a potential problem in both the layered approach to environments as well as the incremental approach. Again, we cannot simply take a series of tools. The tools have to work well together. In order to work well together, they have to know about the underlying database, how the program is stored, and how the data is stored. When you build an environment from scratch, you have a better chance of making things work well together than if you start adding to an existing environment.

We have seen few attempts to use modern workstations and graphics and the full capabilities of modern hardware. This area seems to be lagging behind by several years.

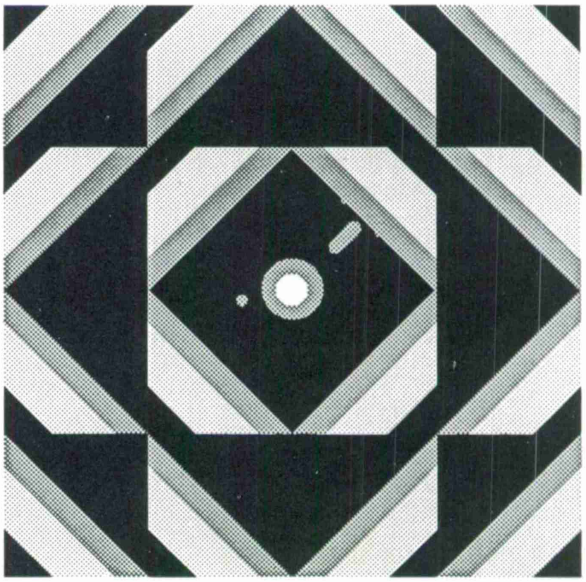
Finally, we have learned that environments are very complex objects that are not only difficult to build, but are difficult to evaluate. We have spent more time than we expected on our evaluations. Validation and design reviews are no guarantee that the system is going to have to have high quality. There are many blatant errors and poor performance in some compilers and environments.

The only way to really get an appreciation for the quality of an environment is to use it in significant ways. That is what we have attempted to do in our evaluations.

I think that you could probably say that the environments that we have today for doing Ada do not have all of the qualities that I have mentioned. I have yet to find an environment that includes production quality and comprehensive functionality and is user friendly as well. I think we are making good progress. I think that much more prototyping has to be done and that we still have to work very hard on defining our requirements for environments. I believe that the incremental environments constitute the best approach at the present time, although the approach of creating an Ada machine looks very promising.

Session 5

Should There Be a New Life Cycle?



Moderator: Christine M. Anderson

Dennis D. Doe

Manager of Engineering Software and Artificial Intelligence
Boeing Aerospace Company

In 1981, Boeing started an initiative to improve our ability to develop embedded mission-critical software. We put together a software standard or methodology at about the same time as the release of DOD-STD-2167.

We released our software standard in late 1982, and then prepared guidebooks to help implement that standard. In parallel with that, we set up a corporate-wide training program and started a project called the Boeing Automated Software Engineering System (BASE), to automate the process we had defined. The Boeing Software Standard (BSWS) 1000, is very similar to DOD-STD-2167 but also gets into the areas of management and operation and maintenance, and has product standards which are equivalent to the 24 Data Item Descriptions (DIDs). Our guidebooks describe how to implement the standard and how to tailor it. As with DOD-STD-2167, we use the software development plan primarily for deciding how we will tailor the standard for a given project.

We have a fairly extensive training program that goes along with the standard, not only for the software engineers and the software managers, but also for the interfacing people and the program managers who helped us implement the standard. We also conducted a requirements review to determine how to automate the process, and we developed a master plan to incorporate the automated software implementation system. The life cycle in our standard is very compatible with DOD-STD-2167, and we are in the process of changing it to become completely compatible with DOD-STD-2167.

BASE is set up to handle several aspects of the program, including management, technical development, maintenance, and automated documentation. Through evolutionary releases we get near-term benefits to our programs.

BASE is a loosely coupled local area network of heterogeneous processors with a variety of operating systems. The software tools in BASE share a common user interface and exchange data through a common database. The basic system contains a mixture of workstations, file-servers, PCs, and terminals. Different projects implement different subsets of BASE hardware and software and consequently get different degrees of capability. For instance, on our Peace Shield program, the implementation of BASE is VAX/VMS with IBM PCs and some terminals. Peace Shield doesn't get the advantage of much of our automated software documentation process, but does get a lot of tools.

As an example, we have a requirement traceability tool on the VAX which we used to do a requirements analysis on Peace Shield. We found that there were about 800 unallocated requirements; if we had let them go until the test phase, they would have been very expensive to fix. These tools are starting to pay off for us.

Our approach was to concentrate on the front end and the integration and test end of the life cycle, where we think it has the most benefit. We are also working to get Ada integrated into the system; Ada is available on the VAX, but we don't have all of the cross-compilers that we need.

We are using Apple Macintoshes right now for our managers, networked together with mail. Our secretaries and many of our engineers are

using Macintoshes as well. We are starting to use some of the tools provided by Apple to do system engineering work: functional flow diagrams, operating sequence diagrams, and so forth.

The concept is one of a common user interface and database approach. We have heterogeneous computers on this system, but we have a database structure setup where, for instance, an automated documentation system can get into the database, pick up the requirements data or whatever it needs to do, and automatically build the documents. The automated documentation system uses the model documents to automatically build specs and the design documents and so on.

Our approach is to use commercially available tools whenever possible. This is not working out as well as we would like. The problem is that since we don't own these tools, we run into a variety of licensing problems when we want to distribute the software within Boeing.

For our requirements analysis work, we are using commercially available tools and/or systems, integrating those tools through a database and providing a common user interface at the workstation. Our prime workstation is the Apollo right now, but because of the cost we are trying to integrate more PCs and Macintoshes.

We are concentrating on the requirements and design phase of BASE now and doing some work on the test end. The work that we are doing for AWACS is a set of tools for automatic system verification. Configuration management, a big part of our system, includes managing the documentation and managing the code. We intend to bring products from the Software Productivity Consortium (SPC) into BASE and then transfer those out into our different software activities.

I will now give you a brief overview of what is going on in the Software Productivity Consortium. We spent the better part of a year putting together a technical development plan and a business plan. We have a five-year plan laid out with the intent to have additional five-year increments as we make progress.

Originally, I thought we would have been up and running by now, but it takes a long time to bring 14 companies together. We are just getting started on the staffing process. The chief executive officer has been selected. The next step is to choose a chief technical officer and then start bringing in people from the companies, from direct hire and from the universities.

Our technical plan has three research thrusts to improve software productivity. These areas are reusable software, prototyping, and the use of knowledge-based software engineering to improve the software life cycle. We hope to get at least an order of magnitude improvement in software productivity. Within the software system engineering portion, we intend to measure our improvements in productivity.

We are a little more near-term oriented than things like the Microelectronics and Computer Technology Corporation (MCC). The companies are signing up for a three-year term in the consortium, and have to give a one-year advance warning if they are planning to get out. We felt we had to have some products in the two-year time frame for them to base their decision on, and we felt that we could provide some tools and methodology in the area of reusable software. Prototyping is a little bit further out, and knowledge engineering is a little bit further out than that. We also plan to look at some future programs.

The system engineering and technology transfer groups within the SPC are the interface between government, industry, and university work, and are intended to bring that technology into the consortium and get it into the program areas. They also interface with the member companies to transfer the products into the member company environment.

I chair a committee that sits between the member companies and the system engineering group that works the needs of the companies back into the consortium. Our needs and reports are reviewed by that committee before they go out to

the member companies, so we are very hopeful that with our plan we can accelerate the technology transfer quite rapidly. By putting some of the better people from our companies into the system engineering area, we are hopeful that we can also accelerate the technology transfer area. We are putting together a software development environment at the consortium, and the member companies that emulate that environment will probably stand the best chance of accelerating their technology transfer as well.

I believe that the existing life cycle has some shortcomings, and I have some suggestions. The existing life cycle is very rigid. We get into a lot of trouble in trying to define our detailed requirements and cast them in concrete before we really have had enough interaction with our customers. We need to do more to provide a support mechanism for prototyping and the evolutionary requirements changes. I don't think the life cycle addresses the operations and maintenance part of the system, and it certainly needs to be flexible enough to support projects of different sizes and types.

I read a paper by Barry Boehm that recommends a spiral life cycle, which has a lot of good features. I'm not sure it's the ultimate answer, but it seems to be heading in the right direction. We have got to support rapid prototyping somehow in order to get our requirements defined and understood by our customers before we go through our PDR and cast our design requirements in concrete.

In summary, I think that DOD-STD-2167 is a good methodology as long as you use a set of guidebooks and do some tailoring to make it fit your given projects. I think for right now that we have at least a foundation to begin from. We can get significant productivity improvement using that life cycle and improving our automation and so on, but if we are going to get real improvement in productivity we have to take a new look at it. We are doing this in the Software Productivity Consortium.

Edward H. Bersoff
President
BTG, Inc.

The recent report to the President by the President's Blue Ribbon Commission on Defense Management discusses increased use of off-the-shelf software, reuse of software, adapting or building new systems only when current systems are clearly inadequate, and also emphasizes the use of prototyping in the development of systems. What it doesn't tell us is exactly how to do that.

The new issue of DOD Directive Number 5000.1 talks about looking at using systems that currently exist, modifying things that exist, and minimizing the time it takes to build a system rather than looking to the frontiers of technology. Minimizing time is a basic emphasis that reusable software and prototyping deal with. Prototyping is again mentioned in the new issue of 5000.1. It also talks about changing things with respect to the standards, and adding test phases or test articles or omitting phases — hardly a bureaucratic view of building systems. It seems to have been written out of frustration with the current process.

SECNAV Instruction 4210.6, Acquisition Policy, published 20 November 1985, talks about making changes only sparingly — don't over specify, avoid unnecessary requirements, use off-the-shelf equipment, and reuse CPCIs or CSCIs wherever possible.

The problem is not so much what we ought to do, but how we ought to do it. How will prototyping and reusable software technologies affect the life cycle in the next year or two? How does the artificial intelligence systems development process affect the life cycle? Artificial intelligence developers seem to be the last haven for people who don't want to deal with the standards and rigor of system development. I think the prototyping and software communities are coming to the realization that current development standards may be counterproductive. Embedded in that issue is the question of whether we should be specifying the *process* of software development or the *products* of the software development. My bias is in the area of product specifications as opposed to process specifications.

If there does need to be a new life cycle, what does it look like and what new standards, if any, need to be developed to support it? We deal with two different kinds of problems in the systems we build. One is the deterministic problem, where there is a sensor input, a response by a computer, and an output process. In such a system, the inputs are well defined and the outputs are fairly well defined. There is also very little human/machine interaction. A weapons system would fit in this category.

The non-deterministic problems, however, are where there is a more intimate human/machine interaction, where the experiences of the user have an impact on how the system gets built. While weapons systems fit neatly into the first case, command, control, communications, and intelligence (C^3I) systems fit into the second. If you look at the case studies that have been presented, the communication system that fits nicely as a deterministic kind of system had very few Software Problem Reports written against it and seemed to be well under control. But the C^3I system that was talked about had 7,500 SPRs during development.

I believe the reason for this is because we don't really know what the problem is at the outset and when we get to the end, the problem has changed, so the system doesn't match the problem anymore. We deal with hard questions in the non-deterministic kind of systems we build. What does this aggregation of forces mean? What happened the last time this particular condition was seen?

The formal life cycle mechanism and DOD-STD-2167 apply very nicely to the first class of problems and less nicely to the second class. I think that much of our problem stems from the application of the rigor of the standards that we currently use in the second class of problems.

Everyone who talks about software engineering speaks of the stages of the development process, from the test planning and the interaction of that activity with the development process, to the delivery and the production/deployment of the system. There is a lot of controversy as to when the life cycle is finished and the characteristics of the operation and maintenance phase of the life cycle. I believe that the operation and maintenance phase of the life cycle is really the life cycle itself; there is no such thing as maintenance. It's just an adaptation and evolution of the system itself.

An Army general at an Armed Forces Communications and Electronics Association (AFCEA) convention mentioned that the project he was working on had a development cycle of 14 years. That is not unusual, but the problem changes over time, and the best we can do is to get to a point where we are not totally mismatched between the problem state and the solution state.

What does that look like in terms of what is really going on, especially in non-deterministic kinds of problems? A need for a system is first recognized (Figure 1). A development process is then begun, but by the time the first article delivery is made at t_1, the user needs have changed, and the first article doesn't even satisfy the requirements that were in place at the outset of the development process.

Impact of the Traditional Life Cycle

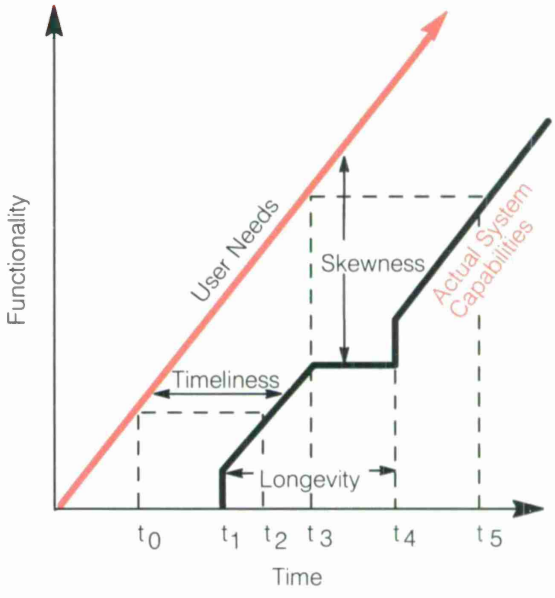


Figure 1

There are some metrics we can look at as to the goodness of the product that we have delivered relative to user needs, because that is really where the productivity rests, with how well we satisfy user need. We want to look at the goodness of the solution relative to the problem.

Adaptability is one of the measures of the system, and is represented by the slope of the line that runs from t_1 to t_3. The timeliness of the system is the time from t_1 back to t_0, which is how far away in time we are from the real problem at any given instant. The skewness is the measure in solution space from the problem. The vertical axis would define the skewness of the solution set.

We adapt a system to a certain point, then realize that the system can no longer be adapted. It ends its maintenance phase. The bugs are too frequent and we have a problem modifying the system, so we end up freezing the system and starting a new system, going through the same process over again. But the problem is that we

are always behind. We need to do something about this, and I think technology can help us to combat our enemy in this whole process, the enemy being time and an understanding of the problem.

One of the most fruitful technologies we can apply is prototyping. Prototyping is a partial implementation of a system used to learn more about a problem or possible solutions to the problem. In the C^3I or non-deterministic environment, the major benefit of prototyping is to learn more about the problem as opposed to learning more about the solution to the problem.

There are several approaches to prototyping, and we need to look at them differently. The simulation or traditional prototyping that has been talked about is either the "throwaway" or "foreign host" approach. The "throwaway" approach is a prototype software system that is constructed on the actual host hardware. This software is usually discarded after the desired knowledge is gained. The "foreign host" approach is a prototype software system that is constructed on different hardware than that which will be used in the actual implementation in order to learn more about the problem or its solution. The actual hardware is often simulated on the prototype hardware. These are very useful in learning more about solutions and, in some cases, problems. But the real benefit lies in the evolvable prototype, which can grow with user needs.

Prototyping does not come free. Often we use prototyping as an excuse for bad design or bad implementation, so we can justify the fact that a system doesn't work by calling it a prototype. In some cases, however, the government likes the prototype and wants to make it production quality. It is generally a mistake to begin with a prototype because it can't necessarily be converted into a production-quality system.

The speed of prototyping doesn't necessarily lend itself to the full rigors of MIL specs and MIL standards, for very good reasons. The DOD wants full life cycle documentation for maintainability, adaptability, and for tracking progress, and 2167 and 2168 embody those ideas. "Throwaway" and "foreign host" prototypes are useful in the early phase or the requirements phase, and sometimes we write code before we finish design in the "throwaway" mode.

Evolvable prototypes, where the intention is to put the quality in after the problem is understood, are discouraged and properly so because you can't build the attributes of quality in after the fact. There doesn't seem to be a mechanism to support the built-in quality approach to evolvable prototyping. That is a major shortcoming in the current life cycle model with the current standards that we have.

Reusable software has been defined by the Software Productivity Consortium as software solutions applied by developers for differing applications within an application domain, such that little or no manual modifications are required. The utility of reusable software is that you get mature solutions and shorter development cycles. Remember, our big enemy is time. All of the prototyping approaches can benefit from reusing software that already exists. Software is comprised of specifications, designs, data, code, test cases and test data, and documentation. I believe that all of these components can be reused.

There are several approaches to reusable software. In the first case, a library of software components is built, and a system is composed in the library by gluing together those components. Right away you see that there is a problem in knowing what is in the library and what its attributes are. There is a whole set of metrics and standards that are absolutely necessary to define what is in the library and to specify what has been pulled out of the library.

Software synthesis would allow one to take a requirement specification which is correct and put it into a tool, and an automatically composed set of modules from the library would be pro-

duced. This approach is obviously much more difficult to implement than the library method.

Software adaptation would be a process by which a piece of software would be adapted on its way out of the library to fit particular needs. An example of this would be a piece of software in the library that was written in the wrong language or written against the wrong specification or standard. If you want to change the precision or the process, you would specify those changes to the system, and the library tool would pull the software out and make the modification for you. Obviously, this is a few years away.

One of the major reasons procuring agencies don't require more reusable software is that they believe there's a specification compromise, that the system is not going to do quite what they wanted it to do, therefore, the reusable software component isn't good enough. If we sometimes over-specify systems, it may seem that the reusable components are not useful, but if we compromise a little bit, we can make good use of things that are in the library and possibly get a system that doesn't quite meet all of our needs, but we can get it sooner.

Another reason for not using the reusable software is standards mismatch. The software was built in accordance with MIL-STD-490 and we want it in accordance with DOD-STD-2167. I believe that the big factor is uncertainty, or lack of control. What is really in this piece of software that we are pulling off the shelf? The contractor who is using reusable software has an excuse for things not working because he was forced to use a reusable component.

There are also reasons for the developer to seek to avoid reuse of software. Lack of control is also a factor for the contractor community. Labor is sales, and if you reuse software, you don't get as many sales. Profit only goes so far. Perhaps it is possible to "incentivize" the use of reusable software through additional profit, but profit is not the same as people and overhead and G&A, and you lose a lot of that if you reuse software. This is another negative aspect to the reuse of software.

Having talked about what the technology is in reusable software and prototyping, let's look at their impact on the life cycle. The "throwaway" or "foreign host" prototypes, in my view, have little impact on the life cycle. Prototypes built in parallel with the requirements analysis and design and even in parallel with the implementation help you do a better job, but they won't necessarily change the basic structure of the life cycle.

However, with evolutionary prototypes, if you have to finish the system development before you learn what needs to be learned from the prototyping, you get a life cycle that is iterative. In some cases, you can learn something about the problem earlier and during the requirements phase you will get one life cycle going and then start another.

When you have multiple prototypes, many things are being developed simultaneously, and this is a potential configuration management nightmare because you have multiple baselines at the same time. However, the good news is that you are higher on the satisfaction scale. Perhaps you get a little bit more timeliness and system deliveries come a little bit earlier, but the big benefit is that you learn more and, therefore, solve more when you build a system.

The evolutionary prototype, in my view, has the biggest payoff because you get some things very early, potentially build in a mechanism for evolution and adaptability, and then rebuild later on. The number of problems you haven't satisfied is much smaller.

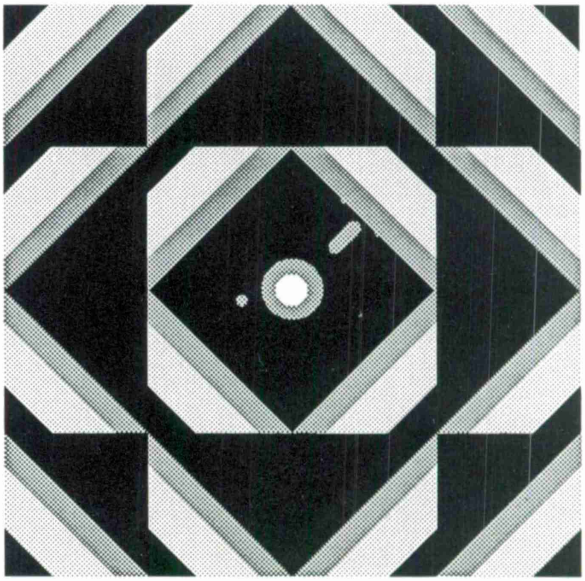
With respect to reusable software, the major benefit seems to be earlier solutions to the problem. The life cycle structure or your solution structure looks about the same, but answers come out earlier.

The answer to our problem takes the life cycle that we understand and, by increasing phases in

scope and shrinking things, tailors that life cycle by making the requirements phase very long, so that we have many documents or things that we want to look at in that period. We then specify what the products are that we are looking for, and adapt the products of the process to a particular procurement. Then we should only buy what we really need and "incentivize" people to exhibit the behavior that we want them to exhibit.

Session 6

Are New Business Practices Needed?



Moderator: Catherine M. Burzik

Pamela Samuelson

Principal Investigator, Software Licensing Project
Software Engineering Institute

Department of Defense (DOD) people as well as industry people tend to agree that the existing DOD regulations on data rights, at least as they apply to software, are too complicated, too long, too ambiguous, not well tailored to the kind of technology that software represents, and are unnecessarily divergent from standard commercial practices. I think new practices in this area are needed if the Defense Department is going to be able to get the best software technology at a reasonable price and to acquire an appropriate set of rights in software.

About a year and a half ago I was blissfully ignorant of the Defense Department's data rights regulations. I was teaching at law school and studying intellectual property law affecting software. I became interested in the kinds of problems that the largest buyer of software in the world, namely the Defense Department, would have in acquiring software.

I started my investigation of software acquisition, as part of the Software Engineering Institute (SEI) contract, by interviewing people involved in the software acquisition licensing and maintenance litigation business: contract officers, procurement personnel, logistics people, lawyers, DOD people from all the services and from the Office of the Secretary of Defense (OSD), and some people that DOD people had recommended. About 120 interviews later, my staff and I went through the regulations and statutes, did some other kinds of legal research, and put together a report. The report examines the range of data rights problems affecting software that were raised by DOD people.

Many people that I interviewed were having trouble understanding what the standard data rights clause that governs DOD's software acquisitions says and means. Many had formed an abstract notion about the meaning of the clause, which was that if the government had funded the software, the government would have unlimited rights in software, and if the software was privately funded, then the government would have limited rights to the software documentation because that is considered technical data under the DOD policy, and restricted rights to machine-readable code which is defined as software under the regulations (Limited rights are government-wide, whereas restricted rights are site-restricted). Others thought that if the government paid for it, the government owned the software, and if private industry paid for it, industry owned it.

That notion is not entirely accurate. Many people thought that "unlimited rights" was a kind of ownership interest. I looked at the definition of unlimited right in the regulations, and it doesn't say anything at all about ownership. It talks about rights to use, duplicate, and disclose, which are very important, but in this area ownership rights tend to be defined in terms of rights to exclude, to control what others can do with the software, not what you yourself can do with it.

Since there is nothing about a right to exclude in the definition of unlimited rights, that was some evidence that maybe unlimited rights wasn't an ownership interest. If the Defense Department wants to own and control software, to have those exclusive rights, it's supposed to use the "special works" clause. That clause purports to give the government a direct ownership interest in software as if it were a work made for hire.

There are a couple of problems with the DOD special works clause. It conflicts with the copyright law in two respects. One is that software is not a category of work that qualifies as a specially commissioned work for hire.

Secondly, and more importantly, the government is prohibited from taking a direct copyright ownership under Section 105 of the copyright law. It may be that the effect of the DOD special works clause is to put the software in the public domain, which is what the Copyright Office seems to think. When it's in the public domain, anyone can do anything they want with it.

The proposed Federal Acquisition Regulation (FAR) and the NASA regulations include a special works provision that might work to give the government ownership interest. In some circumstances, they would permit the government to require the contractor to obtain and assign a copyright to the government. If the government has an ownership interest in software intelleetual property, that would give the government a more extensive set of rights than if the software is in the public domain, because the government would then have the right to exclude others as to the software.

The definition of unlimited rights in the DOD clause includes the right to use, duplicate, and disclose software, but there is no reference to a right to make derivative works. With respect to software, derivative works are particularly important. Modifying software, enhancing it, translating it from one language to another, rehosting it, and retargeting it all depend on being able to make derivative works.

Some people in the DOD say that derivative works must be implicitly included in the definition of unlimited rights, but it's so easy to put a reference to derivative works into the clause if it is what you really want. The proposed FAR defines unlimited rights to include the making of derivative works, which really makes a lot of sense for software.

Another concern about the data rights elause as it's presently drafted is that there is a serious ambiguity in the regulations that has to do with the effect of copyrighting software on the extent of the government's rights. The standard data rights clause allows the contractor to copyright any software developed at public expense unless the special works clause is used. In that same clause, it allows or requires the contractor to give to the government a lieense to use copyrighted software and do various other things with it for *government purposes*.

It may be that the effect of copyrighting a piece of software cuts back the government's rights from unlimited rights to "government-purposes" rights. However, the regulations are unclear on this, and many people seem quite ignorant of it. An ambiguity of this sort can give rise to some serious problems, particularly in the software area.

Let me give you an example of a situation in whieh it matters whether or not the government has true "unlimited rights" or only government-purpose rights. Suppose that the government lets a contract for the development of a software environment to Contractor A, who develops it. The government then contracts with Contractor B to make a derivative program based on that software, and Contractor B wants not only to deliver that rehosted environment to the government, but also to be able to sell it in the commercial arena. If Contractor A copyrights the software that is initially delivered to the government, that may cut back the government's right to a government-purpose license, in whieh case Contractor B's commercial distribution might run afoul of Contractor A's rights. The government really doesn't have the power to give a broader set of rights to Contractor B than they were able to get from Contractor A, especially since derivative software for a commercial market is a very likely possibility. If the government had true unlimited rights, it could authorize Contractor B to sell its derivative of Contractor A's software in the commercial market with no liability to Contractor A. It seems to me that this ambiguity is worth worrying about.

If software is privately funded, the abstraction is that the government will have limited rights to the documentation and restricted rights to the machine-readable version of the program. This is not entirely true. There are actually two different kinds of restricted rights, one pertaining to commercial software and one pertaining to "other than commercial" software or to commercial software that the owner decides to have treated as "other than commercial" software. Those two sets of restricted rights are very similar, but they're not identical. That abstraction is also not true in that the documentation is subjected to restricted rights treatment when the software is commercial and its vendor elects to have it treated as commercial software. Not all software documentation is data to which the government has limited rights.

When a commercial software vendor decides to have the software treated as commercial software, the documentation can be subjected to the same restricted rights rather than limited rights. From the industry standpoint it is very desirable to have a site restriction as to documentation instead of government-wide rights to copy and disclose. This is much closer to the standard commercial practice of treating software and documentation as subject to either the same set of rights or even more restrictive rights as the documentation. The DOD policy tends to reverse that and treat documentation to a much wider set of rights. This is one respect in which it unnecessarily diverges from standard commercial practices.

There are two ways in which things that are privately developed and which seem to qualify for limited or restricted rights treatment can be arguably subject to unlimited rights treatment. My report talks about them at some length.

There are three kinds of flexibilities in the regulations. The government can negotiate up from limited rights treatment in privately developed software. It can also negotiate up from restricted rights to have the software competitively maintained and give out the documentation to someone else. It's also possible to negotiate up from restricted rights for other than commercial software or a commercial software manufacturer whose vendor decided to have it treated as other than commercial software. There is also the option for the commercial software person to opt into the other than commercial software possibility.

However, there is no flexibility to negotiate down, so that it's not possible, if the software is publicly funded, to negotiate for less than unlimited rights without getting a deviation, and it's not possible to go beneath the standard floor of the four restricted rights in code or the standard set of limited rights in documentation. The regulations may not give the government sufficient flexibility at times when it may be necessary, in order to get really good technology. It may sometimes be worthwhile to negotiate away the government's right to modify software in order, for example, to get a warranty on the software.

The proposed FAR has a simpler policy. It provides that, if software is privately funded, both documentation and machine-readable code are subjected to restricted rights treatment, so it avoids all that complexity that is associated with the DOD policy. It also clarifies that the government gets unlimited rights except where the software is copyrighted, in which case it gets government-purpose rights. Perhaps it would be in the Defense Department's best interest to adopt this policy and maybe supplement it to the extent necessary to fulfill special mission needs.

It is possible under the FAR, in situations where both government and industry funds go into a project, for the government to take less than unlimited rights. I think this creates a very significant incentive to get good technology to the government. It also allows the government in some circumstances to take less than the standard rights, for example, to give up the right to modify in order to get a warranty.

Maj. Gen. Henry B. Stelling (USAF/Ret.)
Vice President
Rockwell International

I have a two-part speech. The first part will deal with some software activities at Rockwell International. The second part deals with a 1983 Armed Forces Communications and Electronics Association (AFCEA) study of C^3 system acquisition in which I participated. I will review the evolutionary acquisition approach that the study recommended and I will address some of the controversy surrounding evolutionary acquisition.

In a recent review of the software situation at Rockwell, a consultant found that senior executives were frustrated with software developments. It was suggested that one way to alleviate this frustration would be to conduct training programs. This would enable executives to at least ask the right technical questions rather than accept the software manager's refrain of "Trust me!"

Our systems at Rockwell have tough requirements, and I think that this is true for all of us. These requirements include distributed multiple processors, high availability and reliability goals, and real-time, multiple task environments. Like the rest of industry, our projects are budget-limited and schedule-sensitive. Software is consuming more and more of the budget and always seems to be late.

I believe that many of these problems are really caused by both the zeal of the customers to get approval for programs that they believe are in the best interest of national security, and by the problems that industry faces in a competitive environment. While management metrics are desirable, when we do identify problems in our programs we have little flexibility to correct the schedule and cost imbalances.

We can address these problems by borrowing from the experience of the rest of industry. For instance, during testing of the B1 bomber, we found that the use of computers to assist in diagnosing the problems during aircraft testing is amenable to expert systems. In the integrated diagnostics approach, we borrowed from the expert system approach to give us a much better capability of defining and locating problems in the B1 system. We have the expertise. We use knowledge engineers to come up with a knowledge base, giving ourselves a capability to field an expert system. We use high-order languages, including Ada.

There has been some positive progress over the last decade. Software has been elevated to a level of greater importance. There is a big push to establish a discipline for software in the areas of planning, standardization, communications, and management. We are also using technology to improve our handling of critical missions.

Now I'd like to talk about the evolutionary acquisition approach that was defined in the 1982 AFCEA Command and Control (C^2) System Acquisition Study. I think it is important when we talk about C^3I systems to recognize that we are talking about a set of systems. There shouldn't be any problem in defining hardware and software requirements for sensors, communications systems, and radios for C^3I. We have a unique situation when we deal with battle management of forces, where there is a requirement to support military commanders facing evolving threats. We must accept that battle management requirements will change frequently in the face

of changing threats and Joint Commanders' preferences. In such a situation, agreement on total system requirements can be best achieved — and perhaps can only be achieved — by a building block approach where we build and test the system incrementally.

The evolutionary acquisition concept mandates user involvement in both development and testing as the best way to reach agreement on requirements. It does represent a departure from the Air Force Systems Command's "home base" concept in which the Program Office prefers to stay at Hanscom Air Force Base, in the case of ESD, where all of its support functions are located. However, user involvement is best accomplished by collocation of the System Program Office with the user. It does involve the user in a role that he may not be equipped to handle. In the case of NORAD, they did have a software capability and ESD did set up a team approach to the development of the NORAD Cheyenne Mountain Improvement Program.

The evolutionary approach also stresses the Services' formal requirements validation process. If the evolutionary approach is to be successful, it cannot have long delays in starting successive increments that would be imposed by current procedures. Independent testers responsible for validating a system under development as an evolutionary acquisition may feel compromised by the incremental approach. User involvement in testing and requirement generation for the next system increment is not part of the current independent tester concept.

Budgeting for evolutionary acquisition is particularly difficult. Since we are talking about evolving requirements, we cannot state requirements for a total system in other than a representative fashion. What the AFCEA study proposed is that a representative system be used as the basis for budgeting, with the developer required to design within the approved budget. However, to do this requires a system architecture that can accommodate change.

Experience with evolutionary acquisition, such as RADC's involvement with Constant Watch, and ESD's involvement with OASIS and the NORAD Cheyenne Mountain Complex, indicates that contracting should not be a problem. Since each increment delivered is a stand-alone program, it can be competed if competition is in the best interest of the government. If there is a feeling in some circles in the Department of Defense that each phase of a phased program must be competed, this misunderstanding could be eliminated by assuring that the differences between conventional and evolutionary acquisitions are clearly understood. Contracting will also be easier if software is documented so it can be transferred to a new development team and reused.

In conclusion, I believe that evolutionary acquisition is a business practice that is needed for command and control systems.

Barry W. Boehm
Chief Engineer, Software System Division
TRW Defense Systems Group

Currently, software cost estimation models are at best accurate to within 20 percent of the actual real costs about 70 percent of the time. One of the reasons they are not any better is that the data to which they are calibrated isn't any better. Furthermore, I think the situation is going to get worse before it gets better because of the uncertain effect of such new technologies as rapid prototyping, Very High Level Languages, and Ada.

We need a consistent set of counting rules for Ada programs. Lines of Ada code are put through a "pretty printer" and suddenly you have a program that is twice as large. We need to define and collect data very carefully. One exception to all this pessimism is the ESD Software Management Metrics, which I think are going to lead to some very valuable data in the process of helping ESD manage its software projects.

We need better sizing primitives. Again, we all use lines of code because we haven't found anything better. We have tried function points; they work pretty well on small-to-medium business applications, but they do not work very well on real-time or people-intensive projects.

One of the things that I think was good about the STARS Business Practices Workshop was that it recommended that the Department of Defense (DOD) not standardize on a single cost model. That would freeze technology and it would reduce people's options to calibrate and to get a little bit of parallel triangulation on the problem by using more than one way of costing.

The cost models that are currently around are hard to adapt to new technology. Some of them are pretty good at addressing reusable components. Some of them are just barely getting data that provides some idea of what Ada will do to software costs. Several models are doing pretty well on incremental development. Hardly any of them do very well at costing the impact of using fourth-generation languages or prototyping, either on the cost or the schedule involved in a software development project.

I think the most important thing you can do to get a better cost estimate is to better define the software product that you are going to build. A couple of years ago I ran an experiment where seven teams of people built essentially the same product with the same requirements. At the end of this process, the seven teams produced software in the same elapsed time, but the number of man-hours it took varied by a factor of three. The same inputs and the same outputs were required. One of the requirements was to build a user-friendly interactive interface and a single-user file system. The way people interpreted those requirements really gave you the factor of three in how expensive this was.

Figure 1 is based on a curve from my book, "Software Engineering Economics," with elaboration based on data collected since the book was published. As you proceed through the life cycle and define a concept of operation, a requirements specification, and a design specification, you reduce the variance in the cost estimates. If you are estimating before you have a concept of operation, you haven't pinned down the classes of people and data sources you are supporting, and it shouldn't be surprising that your cost estimates may be off by a factor of two to four in either direction. As you get a concept of operation

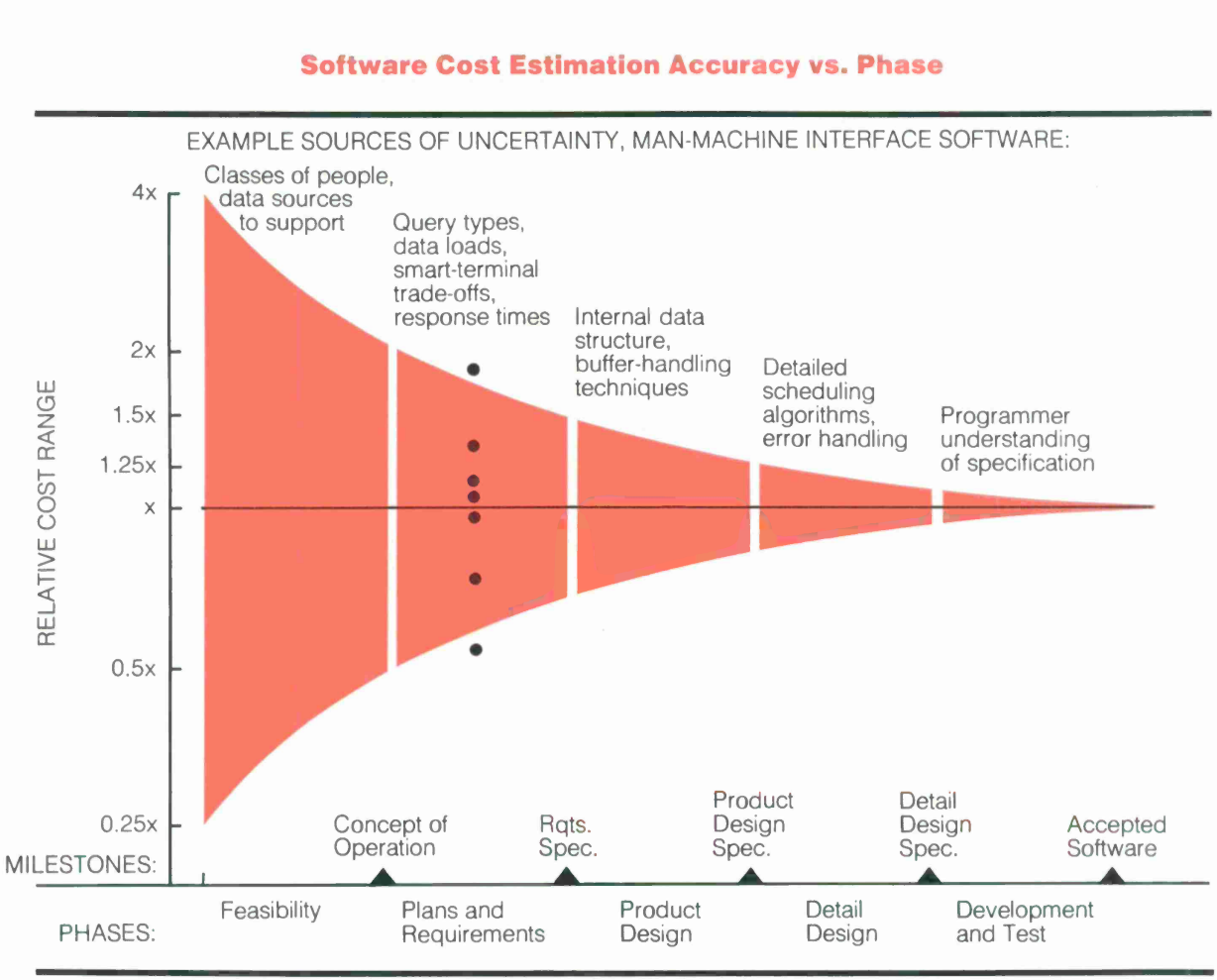


Figure 1: Seven Experimental Programs with the Same Requirements

pinned down, the variance reduces, but at the time of ESD proposals, many other factors that you haven't pinned down still give you a wide source of variation.

In the same experiment, I counted the number of lines of code in each one of these products and found a similar variation: one program had 1,300 instructions, another had 4,600 instructions, and the others were spread in between.

There was one interesting difference that accounted for a lot of the variation in the size and cost. Basically, the seven teams all built programs for COCOMO cost estimation models, and at the beginning of this activity everybody had been furnished the same inputs and outputs, equations, and variables. Some of the other things were very broadly specified: each team was to produce a "user-friendly interface" and a "single-user file system." However, they were all equivalent in that they required user's manuals, maintenance manuals, and well-commented code. They all worked in the same environments.

Four of the teams used a specifying approach. They wrote a requirement specification, they

wrote a design specification, and then they wrote the code. Three of the teams used a prototyping approach. They built a prototype, exercised it, and went on from there to develop the code. Uniformly, the products that went through the requirements and design specifications before writing code had more instructions and were more expensive in the number of man-hours than the products that were developed using prototyping. The average was about 40 percent less code and 40 percent fewer man-hours for the products that were prototyped.

At the end of the experiment, we had three people exercise these programs and rate them on a scale of 1 to 10 with respect to functionality, robustness, ease of use, and ease of learning. The prototyped products were about one point lower in functionality and robustness and one point higher in ease of use and ease of learning. In general, they had less of what people called "gold plating."

I asked the people involved to critique their experience, and one of the specifiers said the big problem with the specifying approach is that "words are cheap." Basically, in doing reviews I would give each team the same kind of feedback. I would say some users would like to put the inputs in backwards as well as forwards. The specifiers would basically view that as one more sentence in the specification, and would put it in. The prototypers, on the other hand, had more of a feel for how expensive that would be because they understood how much breakage there was and how much redesign was involved, so they were much more reluctant to add to the specification.

In using the document-driven DOD-STD-2167 approach to writing down the specifications before you think about the implications, users tend similarly to request all possible options as part of requirements. All of these things get embedded into the requirements and locked into the contract.

On the other hand, prototyping wasn't the universal winner. The specified products had such nice things as interface specifications. The prototypers basically started by dividing responsibility for the software, and four weeks later they would find they were building exactly the same pieces of software that did the error checking on the original and the modified inputs, and they were doing it in incompatible ways. They had specified the same variables with different names and different structures and had trouble integrating them.

From these experiments and similar experiences we have had at TRW, I came to the conclusion that a combination of prototyping and specifying is the best approach, and the best way to determine the mix is by using risk considerations.

Frequently, on both sides of the acquisition activity, we are involved in a situation where RFPs come out at a point in Figure 1 where there is still a factor of more than two for potential variation in the size and cost of the software. One of the things that I think forces us into adversarial relationships is that we lock on to early cost estimates. During the early planning phase of the life cycle, somebody picks a cost and an Initial Operating Capability (IOC) delivery date and your delivery budgets and schedules are fixed for all time. Then we compound the adversarial relationship by coming up with a fixed price contract at a point in time when we really have no idea of what the product is that we are building or what it will cost.

In a situation where our budgets or schedules are unavoidably fixed, I think it's much better to design to cost. There have been a number of procurements that did this in a planned way. The one that I remember the best was the TIPS acquisition at SAMTEC where the requirements specification had a letter in parentheses after each itemized requirement indicating whether the requirement was mandatory or optional, and the bidders could do a proposed design to cost and knew what the priorities of the contract were.

Another problem that I think gets us into lose-lose situations are early Best And Final Offers. What does a Best And Final Offer mean if some of the requirements say "user-friendly interface" or "graceful degradation" or "99 percent reliability" where reliability isn't defined? In reality, they are absolutely baseless and lock us into adversarial situations later on.

The best way to consider the cost variance relationship is that it represents a source of program risk, and the best way to address it is to come up with a risk management plan that addresses the major sources of variation. If we want a user-friendly interface, we can plan to do some prototyping that minimizes the risk. For other sources of risk, we can plan to do more mission modeling analysis, performance modeling analysis, or break the job up into increments where we may better understand what is in increment one. Then we can defer the cost/schedule/performance/functionality trade-offs of the later increments while we get more experience and information.

Another really good approach is a competitive concept definition phase. Basically it takes a couple of competitors down to the PDR so we will have a prototype, a B5 specification, software defined down to the unit level, and cost estimates that are within something about 25 percent of what they would most logically be. In general, we found in the defense software business that if you get the numbers to within 25 percent of what they should be, a good software manager can turn them into a self-fulfilling prophecy. Once you get things pretty firm they are more predictable. If you are a little bit under, the manager can usually motivate people to work a little bit harder and bring the job in on cost and schedule.

If you are going to do the risk management plan, you must actually live up to it. One of the things that we have been working on is the spiral model (Figure 2), which is an attempt to come up with a definition of the software process that is more of a risk-driven process than a document-driven process. One of the problems that we have with the waterfall model or DOD-STD-2167 is that our less experienced people tend to look at it and interpret it literally. If you ask them at any given point what they are doing, they will say, "I'm producing a document." Getting people to focus on the risks that are implied by these documents is very important.

Basically what the spiral model (Figure 2) says is that what we really do, and I think really ought to do in software, is not a linear progression through a sequence of activities, but a cycle at increasing level of detail through a number of processes, the first of which is determining overall mission objectives at the beginning, overall alternatives in terms of centralized or distributed or federated architectures or things like that, and constraints. As you go into more detail, the objectives come down to individual objectives for a little piece of code. Sometimes you will be able to evaluate the alternatives precisely with respect to the objectives and constraints, but generally you will not. If not, you are in a risky situation, so you should go through some kind of a risk resolution activity.

If you don't know what the user interface should be, the model says you ought to do some prototyping. You may continue to do this in each cycle and do the evolutionary development kind of approach. In a situation where you know enough about the job, you can go directly through a concept of operation document, a requirements specification, and a design specification, without having to spend extra time and money on a prototype.

Thus, the waterfall model is a special case of the spiral model, which you use if the pattern of risks in your program say that is the best way to go. Evolutionary development is another special case that you use when the risks determine that is the best way. The spiral model provides a context in which you can use either the waterfall evolutionary development, some other models,

The Spiral Model

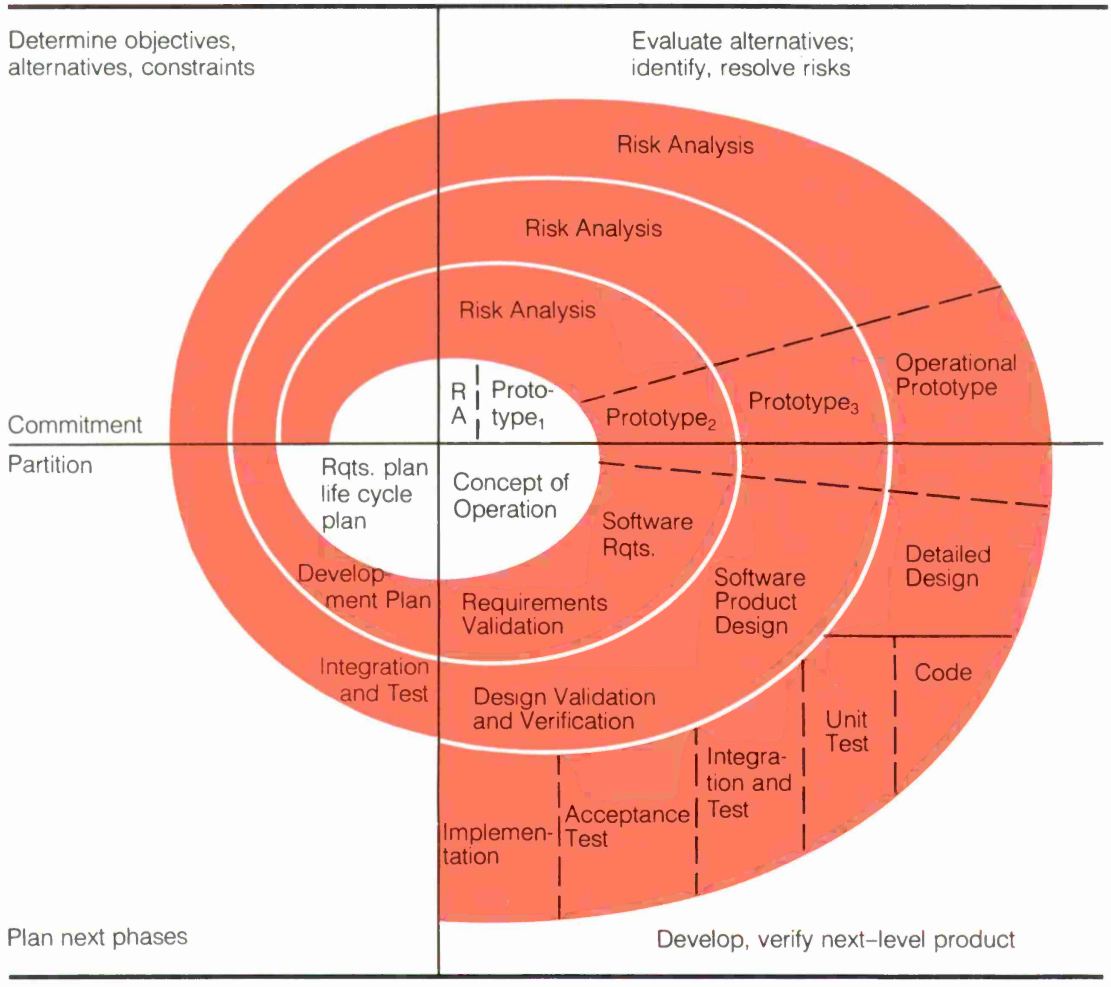


Figure 2

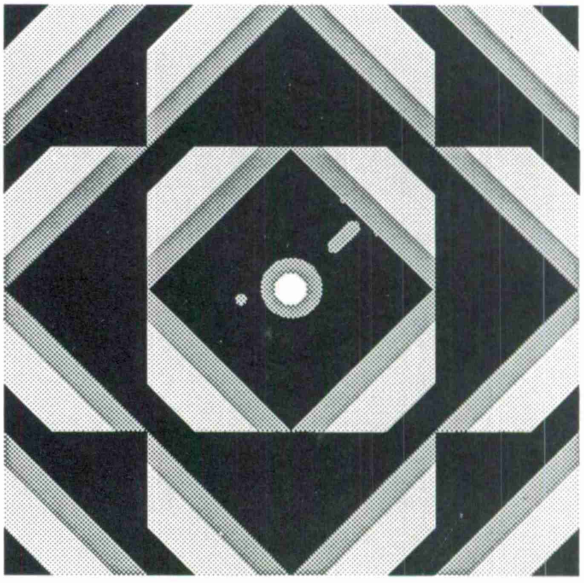
or mixes of these, and use the risk considerations to determine the best mix.

It's a little bit difficult today to go immediately from DOD-STD-2167 to something completely different and not completely worked out like the spiral model. I think there are elaborations to DOD-STD-2167 that involve the specific development of a risk management plan, which can help document the risk considerations and point the acquisition in the right direction. If the risks say we ought to prototype, let's build a plan that gives us enough budget, schedule, and resources to do the prototype and learn the lessons and proceed from there.

Incorporating risk management plans for software is something that we can start doing today. I am encouraged that the upcoming revision of AFR 800-14 includes a requirement for a software risk management plan, and that Air Force programs are already employing them.

Session 7

Where Do We Go from Here?



Moderator: Barry M. Horowitz

Barry M. Horowitz

Senior Vice President and General Manager
The MITRE Corporation

I would like to begin by talking about Ada. A portion of this conference was devoted to Ada, and I feel that a number of things are necessary to move toward its integration into systems. People who propose Ada talk about portability, which implies independence from operating systems and hardware.

We use off-the-shelf software as much as possible, and as a result, many of our systems use commercial database management systems. This has a direct impact on the degree of portability that we have in a given application program. While we may specify the use of Ada on programs, we don't have a specification for portability that we use on programs. Clearly, the above example illustrates that you won't get portability by simply going ahead with Ada.

We talked about Ada and improving the life cycle maintenance of systems, yet I know of no programs, at least that ESD is directly coupled to, where we are helping our maintainers to plan their maintenance environment, not only for the system we are delivering, but for the entire set of systems they will be maintaining. They develop many of their own systems as well as contracting elsewhere.

MITRE and ESD try to do this on a case-by-case basis as these problems emerge, but I think the level of interest in Ada that has been directed down from the Department of Defense (DOD) will continue to rise as compilers become available. However, we don't see the same interest in or even recognition of the need to fund those additional things that will bring the benefits of Ada into the community. I think that all of the people who are interested in Ada and its benefits should be raising the attention of the community of management that funds and supports programs. Maybe this is where the Software Engineering Institute can help.

The second topic I want to talk about is the relationship between the hardware, software, and system engineering groups at companies. I have a unique opportunity in managing MITRE's activities — we work on over 100 Air Force programs, so I see a wide range of companies and how they do business.

My view of the world is that in the command and control (C^2) areas, we are in pretty good shape. The hardware bases are better, and we are using commercial operating systems and database management systems. The concepts for C^2 have not changed radically in the last period of time. As a result, there is an experience base that exists in companies. While there are many software development problems, they are caused primarily by a lack of capacity. However, I think we are going to enter into the next regime in C^2 which will upset that stability.

In the future, people will want to distribute C^2, and go out into vehicles and separate into isolated arenas functions that once sat in the same command post. Communications systems will net these modular capabilities. The hardware and software base will be made so that the modules will be interchangeable, bringing new system engineering into C^2 that hasn't been first order. Much system engineering work will have to be coupled to the hardware and software work.

We're doing worse in the communications and radar area than in C^2 right now, even though the requirements problems are not as acute. This

can be attributed to the explosion in the microprocessor technology. Systems that once did not require software now do. Companies that once didn't have to produce software now do, and they're going through the process of creating an infrastructure like the C^2 people did in the 1960s, with all the associated learning steps. In some cases these companies subcontract, but just as they are not immediately able to manage an internal force, they are not well equipped to manage an external force.

I don't mean to say this is a problem with every single contractor on every job, but I think we are having a very hard time in communications and in the radar area. In the radar and communications area, not only is the processing base getting more and more powerful, but with things like Very High Speed Integrated Circuit (VHSIC) technology coming along, and the demand for fault-tolerant capability, further coupling of hardware and software is going to go up.

One of the problems we see is that the hardware people, the software people, and the systems engineering people in companies are often separate forces. While they are able to bring in a computer scientist who may be knowledgeable about software development environments, this is a long way from bringing in someone who understands how a radar or an anti-jam radio system works. It is difficult to imagine a development force developing the software, which is so integral to the performance of a system, and not understanding the technical functionality. Often, it takes a long time for companies to integrate that knowledge into their software group by bringing these system engineering and hardware people into that group. All the companies should really be thinking quite a bit about how their hardware, software, and systems engineering groups can interrelate, because the demand for that coupling will increase.

Test is the third thing that I want to talk about. There is a lack of trust in the development community — people want to look at software before making production decisions, and examine it in the truest environment possible. ESD, being part of the development hierarchy, is subject to all forms of scrutiny. There is a real need to help people understand that the lead times for hardware and software are different. Only then can we deal with these production decision issues logically.

There is a lack of confidence not only in Congress, but also on the part of the users and the buyers, who think that when a big production decision is made, all the good people leave and a second team comes in to finish the job. I think that great pressure will be put on all of us to ship really good systems. At the time of testing, both the hardware and software will need to be ready.

If we can whittle away at the lack of confidence, then we will be able to control phasing of hardware and software readiness. It will take a large effort for companies and the government to finish a project completely, even if it is late. We must leave time for testing and adequate performance.

Brig. Gen. Michael H. Alexander (USAF/Ret.)

I'm going to talk about some experience I have had in evolutionary development, particularly with the World Wide Military Command and Control System (WWMCCS) Information System (WIS) Program. In developing information systems, which are a little different from embedded systems, often we had to satisfy a user who didn't know what he wanted as we specified the system for procurement, who later told the Operations Test and Evaluation (OT&E) people what he needed, and their tests were run against criteria that were not in our specifications. Working with the user in that OT&E Phase, and with the requirements that the OT&E people buy off on before we can proceed, makes for a very tough problem that we have to continue to work on.

There are some problems in evolutionary development acquisition. In my first program at ESD, we took two aircraft, put the equipment in, flew them, and determined the winner. As a result, we had pretty good performance for getting the competitive prototype.

I have had to go before the Joint Chiefs of Staff and say, "We're doing Block A very well, but, trust us; we'll tell you what we're going to spend the other $835 million on." That is very tough on the budget side, and it's tough on the user too, because he's looking for the total capability, but you're only giving him the first piece. When that starts to slip, you run into problems with overall program schedules, funding, and so forth. Testing is a problem because it is difficult to call the total OT&E community together to test only a piece of what they really want to see, which is an operational concept.

The early phases are very important in evolutionary programming. If you are going to do evolutionary development, be sure that you know up front the total system architecture. Do not sell only the early pieces, promising the rest later. You must have the overall architecture.

Throughout this conference we have been discussing requirements, B-specifications, testing, and other areas with which we have had problems. We have heard about the competitive concept development (CD) phase, which, frankly, I introduced at ESD. We had two or three contractors that thought they knew enough about the system to bid lower. The final cost was lower, but deliveries were usually late. The competitive CD approach has to be done with your eyes open as well.

We have struggled for the last 15 years trying to produce software in the same manner that we acquire hardware. This is true in office automation, and it is true in any of the information systems with which I have been involved. We cannot expect that to work. We have to change the way we do business.

We need to get the engineers, the software people, and the corporate organizations together. This need is not limited to embedded systems or radars, but is true across the board, particularly if we are going to use those people early in the design phase of projects. The whole purpose of Ada is to bring the engineers and the system definition up front early in the program, and to provide the tools to do that from the beginning.

The use of the Ada tools, the Program Design Language (PDL), and the ability to generate the pieces of the system from beginning to end allow you to look at a total system before you sign up to the total architecture.

We are building portability into our Ada projects. As part of the demonstration on every project, we have required that each program show operability on two or more different compilers or in different operating environments. We are well on the way to establishing the Ada standards and our interface standards that are needed to produce the software even before we select the hardware. Our approach requires the hardware vendor to come up with the system that will support our software and our software architecture, not the other way around. Although it is not an official government standard yet, the Database Management System (DBMS) Ada/SQL interface is a major step forward, and we are requiring our contractors to use it in our joint mission hardware selection.

The Software Development and Maintenance Environment (SDME) is not just a conglomeration of tools to help a software developer; it is intended to be a life cycle system. It is an environment for the development and maintenance of all WIS software.

We are not there yet, but clearly we are addressing the issues of portability and software life cycle support, and we are taking major steps to integrate them into an active, ongoing program.

William L. Sweet

Associate Director for Technology Transition and Training
Software Engineering Institute

I would like to introduce the effort of the Software Engineering Institute (SEI) to address the problem of software acquisition. The Software Engineering Institute is a Federally-Funded Research and Development Center associated with Carnegie-Mellon University (CMU) that is not part of industry or government, and is semi-autonomous from CMU.

In this unique place, the SEI is able to address problems that are bogged down in acquisition policy between government and industry, or in the competitive posture between industry elements, or in other isolation between industry and universities. Those traditional barriers have their place, but they contributed to a very long time delay between concepts and practice in the area of new technologies.

In the case of software technologies, already we can find engineering technologies that are 10 to 15 years ahead of where we now are in our practice. We have a lot to gain by accelerating the transition of these concepts into practice.

That is where the Software Engineering Institute enters. The SEI is not intended to be a fountain of all knowledge on software engineering, nor a large research effort on software engineering, but it is intended to be a channel through which the latest software engineering technologies can be transmitted to other places and put into practice. Spreading this use of the latest technologies can be very beneficial to us as a nation, and certainly to the Department of Defense (DOD) contracting community. That is what the SEI intends to accomplish.

Can we as people change the ways we are doing our software development and the way we are doing our acquisition? Maybe what we have been doing is not appropriate for software. Clearly, hardware and procurement practices for hardware came out of the industrial era. As we enter the information age, we have to think about new directions. The assumption that we should use the same ground rules for software acquisition as are used for hardware has proven fallacious.

I think we have to ask ourselves what we need to change, how we should change it, and where we go from here. We need to move on the problem. I think ESD is very enlightened, not only to engage the assistance of MITRE and gain that added technical strength, but also to co-sponsor this symposium. I think this is a very important step on the way to the solution.

It is important for us to remember that the net intent of the acquisition process regarding software is not to provide application code to the Air Force, but to provide operational capability on time. The issue of how we get those lines of code in there should not be viewed as an end in itself; we need to rapidly do what is necessary to bring the lines of code into successful operational use.

Ultimate efficiency in developing software might suggest that we should have hard and fast requirement specifications and be very rigid in our approach to developing the software, but in reality what we want is an operational capability that meets the need. Looking at the flexibility in software requirements as an onerous problem in software development is missing an important point. The very fact that we can use techniques

such as rapid prototyping to enable us to adapt our requirements as we go along much more easily than we ever could in hardware is a potential plus. This enables us to provide systems that more nearly meet the needs of the real users. It is important to focus on how we can make use of new technologies to accommodate changes in requirements. I think we should anticipate and use the changes in requirements in a positive way to better achieve the real intent.

Tony Salvucci talked about the use of oral exams as a way of eliminating some bidders — asking questions about bidders' capabilities in the area of software engineering, rather than deciding the competitive procurement on the basis of the proposal alone. I believe that there are parts of the procurement process that do not fit into the classical competitive proposal process, and that one of those is the selection of capable software development teams. If we were to establish a strong "oral exam" process whereby bidders or potential bidders knew that a very great impact would result if they were unable to present a capable team, we would see a much greater emphasis on training coders into computer scientists and computer scientists into software engineers.

As a former member of the defense contractor community, I'm well aware that the investment in this upgrading and internal professional training of people into higher degrees of professionalism is quite small compared to the effort that goes into proposals. If sole-source contracts were being issued, not on the classical bases, but on the basis of having excelled in these oral exams, I think we would find a marked change in the way we are able to perform on our software.

I would just like to conclude by saying that I think what is needed is courage in contracting. There are opportunities out there. We recognize that software is an extreme challenge to us, and we should meet the challenge by being exceptional in the manner in which we do our acquisition.

John B. Munson
Vice President
System Development Corporation

I was asked to summarize what I heard the last two days, but first I want to admit that I was the chairman of one of the studies that have been referred to earlier as a waste of time and effort. I would really have to agree with that opinion, considering some of the frustration I went through. The software study that we did for the Air Force's Scientific Advisory Board three or four years ago dealt with risk management, cost uncertainty, the need for development tools, investment in the future, and solving the operations and support problem; these are all still open issues that we have again discussed at this symposium.

We did try one different approach. Rather than making a set of free-standing technological recommendations, we tried to turn this around and frame them in an institutionalizing set of actions. This was an attempt to transfer responsibility for these recommendations to the Air Force, as opposed to simply telling them what to do.

I think it is interesting that, in general, the software problem is not just an ESD problem in the Air Force; it's corporate-wide — every activity, every division has "a software problem." It turns out that of all the people and organizations that we had talked to, from the Chief of Staff of the Air Force on down, the only organization that showed any apparent interest in our study was ESD. I'm very pleased to see that today, some of the ideas from our study are being implemented in ESD's procurement practices. That's extremely encouraging to me.

I have been very impressed with the content of this symposium, especially the consistency of the opinion. Generally, there has been very little controversy, meaning we are in general agreement. During one break I talked to some people who felt there was a strong industry-versus-government bias here, but I didn't feel that. I thought there was much more commonality than difference of opinion. I think we recognize the problem clearly, and we can work toward solving it.

The one thing we have all recognized at this point is that the problem is very complex. If it were not, it would have been solved a long time ago. When building software or intellectual systems like software, developing the system functionality is just plain hard work. When you forget that, when you try to look for solutions that don't involve hard work, I think you get led down the wrong track.

There is absolutely no technical or management substitute for understanding the problem. Many times our acquisition process forgets that. The problem isn't just the software engineering problem; it's equally involved with the application problem that we are dealing with. If there is one thing we've learned, it is that the second iteration of a problem always seems to go better than the first, and the third iteration goes much better than the second. There is a learning curve, but the learning curve is more on the application aspect of things, and understanding the problem with which we are dealing. When the software people clearly understand the applications that they are trying to work in, it makes a big difference.

I think if you look back on some of the success stories, as opposed to the disasters, you'll find most of the success stories are related to the fact

that the people who were building the application understood the technology they were dealing with, in addition to being good software people. So, there is no ultimate "software" solution.

I believe that the people who talked about life cycles got the cart before the horse in many respects. Life cycles shouldn't drive the processes. The process should drive the life cycle. Life cycles are around to implement the tools we have to use, and they are the best ways we have today of doing it. The software engineering process is basically a continuous process. The life cycle makes it artificially discontinuous. Our ultimate goal should be to take all of that discontinuity out of the life cycle and go from requirements to application with no stops in between. Since we can't do that today, we put discontinuities in to provide us with visibility and control.

We must be careful not to get locked in to an essentially artificial process and not recognize new ways of doing business. New tools have implicit life cycles in them. Dr. Boehm's discussion of a spiral life cycle tended to emphasize that it's the problem you are solving that drives the life cycle, not the other way around. We must remember that and keep that in mind as we search for new solutions.

While I conceded that this symposium seemed to have reached a consensus, what still scares me about this whole process is that I am absolutely convinced we have finally gotten the technology and the tools to build the systems of the 1960s. If we want to go back and build 1960 systems, we probably know how to do it now. However, today we are into a whole new world of applications where the references in DOD-STD-2167 and Ada say "to be determined" — distributed systems, federated systems, and multiprocessors. But maybe that isn't too bad. Maybe we are at least at a place where we have a baseline; where we ought to start trying to do something instead of just talking about it.

One of the recommendations that I would have, as a result of listening to this discussion, is that we try to take the talking we've done and the agreement we have, and try to turn these recommendations into action items. We should make somebody accountable for putting them into terms of how to implement them.

As a program manager, I find that there are many people who can tell you why you can't do something, and very few of them who worry about trying to tell you how you can do it, and then help you do it. I think that is one of the things we could do to help ESD; we should try to create an action program as opposed to telling them why they can't do it. We may be ready for that.

We should convert our ideas into a set of action items and see if we can make progress against them. I am as convinced now as I was when the Air Force studies were on, that unless somebody takes responsibility and accountability for making these things happen, they will never happen.

Bureaucracy exists to make sure that nobody is responsible for anything, so that the blame can be spread over a large number of people and never focused and isolated. I hope that we as a group don't continue to behave like a bureaucracy. I would like to see us take responsibility for our suggestions and exhibit the courage of our convictions. My recommendation is that we stop talking about it and try to act on some of these items of consensus and see if we can't make some real progress. Doing anything, no matter how little, is better than doing nothing.

Barry W. Boehm

Chief Engineer, Software System Division
TRW Defense Systems Group

I would like to discuss my experience so far on the Defense Science Board Task Force on Software. I can't really talk about the recommendations until they have been presented and approved by the Defense Science Board, but I will cover some personal impressions to date based on the briefings we've received on Ada, STARS, people, and data acquisition rights.

The general impression of Ada is that it is not perfect, but it's better than any alternative that the Department of Defense (DOD) has right now. It's mature enough to be required on new procurements, but there should be an option for exceptions. These exceptions should have a strong rationale, rather than simply being a way around the use of Ada.

I believe that the STARS program is really essential, and will help in consolidating the state of the art, along with agencies such as the Software Engineering Institute. The STARS and SEI can bring us all up to a reasonably strong level of capability.

We have looked at studies that say the Services are short on key people needed to manage acquisitions and software, and that incentives for retention should be provided to stimulate career paths. What has happened as a result of those studies is that people have acted as if the recommendations were going to be implemented and haven't done anything to compensate for the shortage of people, but nobody has ever implemented those recommendations. Career paths for software people in the Services are just as grim right now as they have ever been, and are just as attractive outside the Services.

My impression is that it's better to recognize that we are never going to solve this problem, and to concentrate on compensating for it. The DOD would be better off to redeploy the scarce people they have, and have fewer of their software experts programming and more of their software experts managing acquisitions. These people should be supplemented with Federal Contract Research Centers (FCRCs), SETA contractors, or Independent Validation and Verification (IV&V) agents, so there will be enough people to make sure the software acquisition goes right.

With regard to rights in data, DOD really needs a clear, simple regulation that doesn't require a lawyer to interpret and can be tailored by somebody in an acquisition organization. A customer's data rights needs should be determined by the life cycle plan. Currently, many things are being asked for and nobody knows why; it's a more-is-better kind of phenomenon. People ask for unlimited rights, but they have no idea how the maintenance will be distributed among the customer, the user, and the contractor. The degree of rights requested has not been related to the need for the data or software being asked for.

In terms of acquisition, there was a very strong consensus at this symposium that the most important thing to do is to focus on getting the requirements right before locking yourself into some particular acquisition. There were a couple of comments that are really just codified common sense; you ought to get close to the customers and really have an understanding of what the

users' needs are before going out and trying to build something for them.

Another interesting concept is that of simultaneous loose-tight properties. We feel that DOD-STD-2167 does not encourage a simultaneous loose-tight property. It's too easy to over-interpret into a tight-tight property and get into document-driven acquisition. If you take a risk management point of view, it gives you more of an opportunity to figure out where there should be looseness and where there should be tightness.

Current initiatives like the revised 5000-29, revised 800-14, and the initiatives toward DOD-STD-2167A, appear to be going in the right direction. ESD doesn't really have to wait for all of those regulations to come out before doing something.

I think that others were right in saying that the most important thing is to handle this problem of pre-full-scale-development engineering, and build that into the strategy for acquisition — including the necessary budgets and schedules. I have been really impressed with the initiative that ESD has shown recently in projects such as the red teams, the orals, and the metrics. You don't see very many government organizations coming out with that many new initiatives that seem to be directed toward the right target. The track record of initiative here is very encouraging.

A. Paul Arieti
Vice President
Grumman Melbourne Systems Division

We have been critically examining ourselves for the last two days, and I would just like to offer that despite our problems, I think we have developed some pretty powerful systems. We also have many good methodologies; we just have to bring them up to speed with the technology. We have a very fast moving target.

We have computers sitting on desks that 10 years ago were in a pristine computer environment taking 10 times the floor space. Trying to make use of the capacities that are in the systems now, with the software methodologies we have, is not an easy job. Our situation is analogous to diving competitions. There is a measure for performance, and there is a measure for degree of difficulty. I think that in our industry we get the highest score for degree of difficulty.

Our problem is that the performance is often too poor to be made up by the high degree of difficulty. We have to lower our degree of difficulty somewhat to greatly improve performance. We have to spend more time on requirements, and we have to work them more completely. We have to put more emphasis on requirements prior to Preliminary Design Review (PDR). That means that there must be agreement on use of evolutionary programs, more prototyping, and developing systems and fielding them to the user as part of the development process.

Many people spoke on the necessity of getting the right resources and keeping those resources consistently on the job. More management attention was another theme for many of the speakers; getting the managers in early and making sure they pay attention to what is going on throughout the process. There was also a lot of talk about changing the form of contract or possibly dealing a little bit differently with cost versus fixed-price during the various phases of the procurement of a system.

I favor two-phase procurements to minimize "degree of difficulty." There is a procurement going through ESD right now that is going to require of two contractors the B-specifications, the software development plan, and an engineering mock-up of the total configuration, all prior to the PDR. One contractor will then be selected to implement the system. I think this two-step procurement process can be very effective in alleviating many of our problems.

One of the things that we have to do is accelerate the procurement process. We spend too much time trying to get it right with that first A-specification; an award is then given to one contractor with a five-year span for system development. There is no way that the initial A-specification can accurately reflect the correct need five to seven years after it was written. I favor evolutionary procurement and the ability during the contract definition (CD) phase to take another look at the requirement so that we can be more effective in developing what the user needs. We can better ensure that what we develop and deliver to the customer is going to be a system that is more acceptable to the user.

I would now like to talk about the review process. We can be criticized for not reviewing ourselves enough, but on the other hand, there is often too much review by the outside customers

— MITRE, ESD, and the user community. I think the two-phase procurement approach would also reduce the amount of in-process review needed. The break between the first and second phases would give ample insight into the design.

I recommend that we look hard at the two-step process in full-scale development and include in the first phase an engineering prototype that gets evaluated by the user community. Some of the people I have talked to about this concept believe that it will take longer and cost more. I believe that if we can accelerate the initial procurement process, we can go through the whole process faster, spend less money (even with two awards in the first phase), and field a better production system.

Maj. Gen. Thomas C. Brandt

Vice Commander
Electronic Systems Division

There have been many achievements that are remarkable if you step back a couple of decades. That doesn't mean that tough, systemic problems don't remain; a cultural change may be required to allow us to overcome those seemingly intractable problems or challenges that we will be faced with in the years ahead. At the same time, I think the record of man indicates that when he brings forth his best abilities and his best thinking, he is clever enough to meet any challenge.

We run into major problems when we get trapped in our culture — in our thought patterns — and fall victim to error. When we impute a causal relation to a sequence of things, we get into trouble. We might be trapped in larger institutions by shoddy thinking, which I think we have an opportunity to overcome.

I want to talk about how we should evolve in software and where we should go from here. I believe that we should evolve very slowly. We in the government must act better and think better, and more cleverly in our acquisitions, and industry must respond in kind. We must become more effective and more efficient. We must nurture the evolution of a profession.

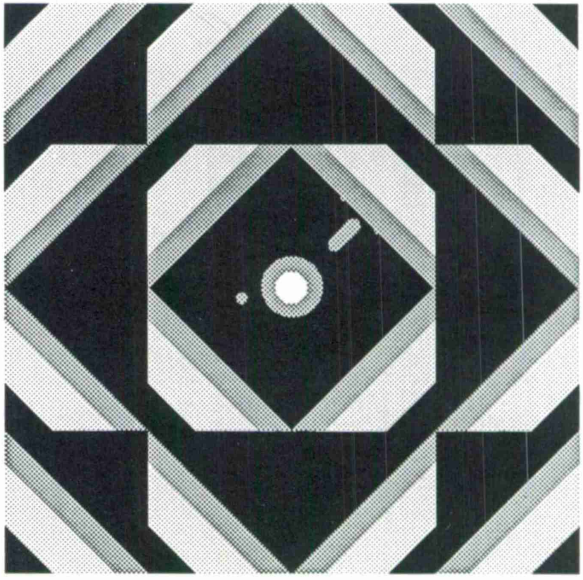
Software engineering today is not a profession, because it doesn't have attached to it all of the critical elements that come about when a discipline evolves into a profession. Coders do not make a profession, programmers do not make a profession, nor do computer scientists make a profession. What makes a profession is a basis in theory and science that then collects empirical data in large amounts over time. Therefore, a collective history is built, as well as understanding, and that process usually takes a relatively long period of time. The experience of failure allows you to go on to success afterward.

We need to think about concepts such as sanctioned standards of quality, codes of ethics, metrics of evaluation, education and training standards, certification, and perhaps ultimately, within our culture, a more well-understood path to the top in industry.

We have done a lot of talking, and dialogue is good. Although we are only describing the problem and no one is forthcoming with solutions, half of a solution is a precise understanding of the problem. As we begin to define, describe, and examine the problem, the clarity of the solution will be forthcoming.

It's not the best of times, but it's not the worst of times either. We keep saying that if there really is a saving grace for our society, it's our people with that intellect, that spirit of entrepreneurship, that willingness to create and discover and do. We recognize what is to be done. The time to do it is now.

Speaker Biographies



Biographies

Brigadier General Michael H. Alexander

Brigadier General Michael Alexander recently retired from the Air Force after serving as the Joint Program Manager for the World Wide Military Command and Control System (WWMCCS) Information System (WIS) in the Organization of the Joint Chiefs of Staff. At the same time, he was the assistant for WIS, Deputy Chief of Staff, Research, and Acquisition, Headquarters, U.S. Air Force.

Prior to his assignment to WIS, General Alexander held positions at ESD, including Deputy for Strategic Systems, Director of the Tactical Long Range Navigation Systems Office, Assistant Deputy for Command and Control Systems, Deputy for Iranian Programs, and Deputy for Development Plans. He has also been commander of the Arnold Engineering Development Center.

A. Paul Arieti

Paul Arieti is Vice President of Advanced Systems in the Data Systems Division of the Grumman Corporation. In February of 1986, he was permanently assigned to the Grumman Melbourne Systems Division to provide senior verification and validation of the software development activity for the Joint STARS Program. Prior to this assignment, Mr. Arieti was head of Advanced Programs in the Data Systems Division. This department provided management of large data systems programs for new business initiatives.

Mr. Arieti has been with Grumman for over 20 years. He has managed the Grumman Automated Telemetry Station and formed what is now the Systems Maintenance Services division of Grumman.

Ernest C. Bauder

Ernest Bauder is Manager of Air Force Systems Engineering in GTE's Communication Systems Division. He is the NSIA Software Committee representative on CODSIA Task Group 21-83, which was responsible for industry review of DOD-STD-2167, the Defense System Software Development Standard. He was also a member of the NSIA Software Task Force for Air Force C^3I Applications.

During his 28 years at GTE, Mr. Bauder's assignments have included Manager of Design Engineering, Assistant Director of Engineering for Software, and Manager of TTC-39 Software Engineering.

Leonard W. Beck

Leonard Beck is group vice president and manager of the software engineering division at Hughes Aircraft Company's Ground Systems Group in Fullerton, California.

The software engineering division is responsible for all application and product line software activities. The group's primary business interests include air defense systems, communications systems, ground radars, shipboard electronic systems, and military displays. Mr. Beck has been with Hughes for over 30 years.

Edward H. Bersoff

Dr. Edward Bersoff is President and founder of BTG, Inc., a high-technology, Virginia-based systems analysis and engineering firm involved in the development of computer-based systems for the defense and civil sectors. Recently, Dr. Bersoff and BTG led in the preparation of the Technical Development Plan for the Software Productivity Consortium, an association of 14 of the largest aerospace defense contractors. BTG's

microprocessor-based Prototype Ocean Surveillance Terminal (POST) employs modern prototyping methodologies and is currently being deployed to over 50 Navy ships and shore installations.

Prior to his contributions to BTG, Dr. Bersoff was President of CTEC, Inc., where he directed the company's research in software engineering, product assurance, and software management.

Barry W. Boehm

Dr. Barry Boehm is currently Chief Engineer of TRW's Software and Information Systems Division. His responsibilities include direction of TRW's internal software R&D program, of contract software technology projects, of the TRW software development policy and standards program, of the TRW Software Cost Methodology Program, and of the TRW Software Productivity System, an advanced software engineering support environment.

Dr. Boehm is currently a Visiting Professor of Computer Science at UCLA and serves on the Governing Board of the IEEE Computer Society. His book, *Software Engineering Economics*, was published by Prentice-Hall in September 1981.

Major General Thomas C. Brandt

Major General Thomas Brandt has been the Vice Commander of the Electronic Systems Division of the Air Force Systems Command since January 1986. He joined ESD after serving as director of the joint planning staff for space, Office of the Joint Chiefs of Staff.

From 1979 to 1984 he was assigned to Headquarters North American Defense Command where he served as assistant deputy chief of staff for space operations; director of space and missile warning operations; assistant deputy chief of staff, operations for combat operations; the first Space Command assistant deputy chief of staff, operations for combat operations; and deputy chief of staff, intelligence, for the U.S. Air Force Space Command.

Delbert D. DeForest

Delbert DeForest is Associate Department Head of Radar and C^3 Software at the MITRE Corporation. Mr. DeForest joined the MITRE Software Center as a Group Leader in October 1985 after 22 years of experience in the development of real-time embedded computer systems. At MITRE, he has been responsible for the applications of Software Reporting Metrics to ESD programs and software acquisition support to the North Atlantic Defense System and the E-3A (AWACS) program.

Prior to joining MITRE, Mr. DeForest held a variety of management and senior staff positions at Raytheon's Submarine Signal Division, including manager of the Software Development Laboratory where he was responsible for the development of software at the Submarine Signal Division.

Jack R. Distaso

Jack Distaso is Assistant General Manager of the Systems Engineering and Development Division in the TRW Defense Systems Group. His organization develops advanced systems requiring the technologies of networks, data processing and communications, artificial intelligence, distributed databases, and fault tolerant architectures. The Systems Engineering and Development Division is primarily involved in military command and control systems, sensor processing systems, weapons systems, management information systems, and communications systems.

Mr. Distaso has been with TRW for over 20 years. He has been manager of a large ballistic missile defense program. He also served as a project manager for several projects developing real-time software for ground and on-board missile systems.

Dennis D. Doe

Dennis Doe is Manager of Engineering Software and Artificial Intelligence at the Boeing Aerospace Company. In this capacity, Mr. Doe is the focal point for software methodology and automation, artificial intelligence applications, and advanced software research for aerospace products. He is also involved in Boeing's Software Automation program, their Software Standards and Guidelines program, their Artificial Intelligence program, and their Ada program.

During the past two years, Mr. Doe has been the leader of the technical group for the Software Productivity Consortium, an organization involving 14 major aerospace contractors. The focus of the consortium is on significant improvements in software productivity and quality through advances in methods, techniques, and tools.

For over 27 years, Mr. Doe has served the Boeing Company in many capacities, including software and systems engineering assignments on the Lunar Orbiter, the Short Range Attack Missile, and the B1 Avionics program.

Barry M. Horowitz

Dr. Barry Horowitz is Group Vice President and General Manager of the C^3I Group for Air Force Systems at the MITRE Corporation, located in Bedford, Massachusetts. In this position, Dr. Horowitz is responsible for MITRE's ESD work program. Prior to this assignment, Dr. Horowitz was senior vice president and general manager of the Bedford C^3I Division at MITRE.

Dr. Horowitz joined MITRE in 1969 as a member of the technical staff in the Air Transportation Systems Division in Washington. Transferring to MITRE-Bedford in 1979, Dr. Horowitz began working on Air Force-sponsored activities. During the past seven years, he has played a key role in MITRE's support for ESD. He was previously the Bedford Division vice president for programs, where he was responsible for all of MITRE-Bedford's planning and acquisition programs.

R. Blake Ireland

Blake Ireland established and has since headed the Software Systems Laboratory in Raytheon Company's Equipment Division. The laboratory is responsible for the majority of the Equipment Division's software engineering.

Mr. Ireland has been associated with software development for military and government systems for over 30 years. Prior to joining Raytheon, he was with the System Development Corporation and the RAND Corporation. He has been involved in major programs such as SAGE, NORAD COC, the Apollo Program, and COBRA DANE.

Robert J. Kohler

Robert Kohler is President of ESC, Inc., a subsidiary of TRW. Prior to assuming this position, Mr. Kohler was Vice President for Advanced Programs and Development in the Space Systems Division of Lockheed Missiles and Space Company.

Mr. Kohler spent 18 years at the Central Intelligence Agency, where he was responsible for the development, engineering, and operation of sophisticated technical collection systems. Most of his work had been with the Directorate of Science and Technology, where his positions included Director of the Office of Development and Engineering, and other major assignments.

Lieutenant Colonel William E. Koss

Lieutenant Colonel Edward Koss is currently assigned to ESD as the System Program Director for Granite Sentry. Prior to this assignment, he was the Deputy Program Manager for Logistics on the WWMCCS Information System program. Col. Koss spent four years as the Director of Space Computer Resources at the Air Force Space Division. He has 11 years of experience in a wide range of space programs including the Anti-Satellite Program, the Space Nuclear Detection Program, MILSTAR, and missions associated with the first Atlantis space flight.

Lieutenant Colonel Koss has a B.S. in Mathematics, an MBA, and a Ph.D. in Finance. He has published 51 articles and four books on the subjects of Computer Management, International Affairs, and Finance. He currently serves as technical consultant and expert witness for the IRS in court cases on computer leases, and as expert witness for the Air Force Judge Advocate.

Charles W. McKay

Dr. Charles McKay is Technical Director of the Joint NASA Johnson Space Center/University of Houston at Clear Lake Ada Programming Support Environment Beta Test Site.

In addition to his full professorship and responsibilities teaching courses in Software Engineering, Control Systems, and Electronics at the University of Clear Lake, he holds the title of the first Director of the University's High Technologies Laboratory, a newly formed organization dedicated to research, conferences, and institutes in the high technologies. Private industry, government, and academia have benefited from Dr. McKay's consulting expertise in the areas of computers and computer automation.

John B. Munson

Jack Munson is currently Vice President and General Manager of the SDC Space Transportation System Operations Contract. This is an 800-percent contract to Rockwell International to maintain all ground-based software at Johnson Space Center for support of space shuttle operations.

During Jack's 30-year career with SDC, he has managed the development of software systems primarily for large real-time military computer applications. In 1984 he led an Air Force Scientific Advisory Board study on "dealing with the high cost and risk of embedded computer software." He is on the Executive Board of the IEEE Software Engineering Technical Committee and a member of the Air Force Scientific Advisory Board.

Gerald E. Pasek

Gerald Pasek is the MILSTAR Mission Control Program Manager for Lockheed Missiles and Space Company. In this position, he is directing the efforts of several hundred technical personnel to provide survivable command and control for the MILSTAR Communication System.

Mr. Pasek has more than 25 years of experience in the conceptual design, development, and project management of large DOD systems. He has specialized in ground support and processing for satellite systems which are typically software intensive.

Alan J. Roberts

Alan Roberts is Senior Vice President and General Manager of the MITRE Corporation's Washington C^3I Division. In this capacity, Mr. Roberts is responsible for corporate management of national security activities and defense systems activities. Mr. Roberts was vice president for strategic systems for MITRE's Bedford Operations before taking charge of the Washington C^3I programs in 1984.

Mr. Roberts holds a B.S. and an M.S. in Electrical Engineering from Massachusetts Institute of Technology. As a research assistant at M.I.T.'s Digital Computer Lab, he worked on the Whirlwind I computer. He was responsible for operation and maintenance of electrostatic storage, and he assisted in the installation of the first magnetic core memory.

Anthony D. Salvucci

Anthony Salvucci is the Assistant Deputy Commander for Strategic Systems at the Electronic Systems Division. He assumed this position after serving as assistant for systems acquisition for ESD's strategic programs.

During more than 25 years at ESD, Mr. Salvucci has had extensive experience in systems acquisition. He was executive manager for the

acquisition of all Tactical Warning/Attack Assessment programs for the Air Force Systems Command. He was also director of the NORAD Cheyenne Mountain Complex Improvement Program. In this position, he was responsible for the complete replacement of the communications, data processing, and display systems for NORAD's air, space, and missile warning command centers.

Pamela Samuelson

Since January 1985, Pamela Samuelson has been the Principal Investigator of the Software Licensing Project at the Software Engineering Institute at Carnegie-Mellon University. She has written an extensive report on the Defense Department's software acquisition policy which recommends substantial changes in DOD data rights regulations affecting software. She has also written a report on how DOD could improve its planning for maintenance and enhancement of software.

Ms. Samuelson is an Associate Professor of Law at the University of Pittsburgh School of Law, specializing in intellectual property law affecting new technologies, antitrust, and broadcast regulation. She is the author of numerous articles on software legal protection.

Major General Henry B. Stelling

Henry Stelling is a Vice President and Director of the Defense Electronics Operations' Advanced Develoment Center at Rockwell International.

Prior to assuming this post, Mr. Stelling was a Major General in the Air Force. His last assignment was as Vice Commander of the Electronic Systems Division. During his military career, General Stelling held many important assignments with the Armed Forces Special Weapons Project and the Directorate of Special Weapons at Tactical Air Command Headquarters, as Director of Space in the Office of the Deputy Chief of Staff for Research and Development, Headquarters U.S. Air Force.

William L. Sweet

William Sweet is the Associate Director for Technology Transition and Training at the Software Engineering Institute. In this capacity, Mr. Sweet is responsible for the acquisition and refinement of software engineering technology and for facilitating the transfer of the best available technology into widespread use in the software-related organizations of U.S. industries.

Prior to joining the SEI, Mr. Sweet was the Division Chief Engineer of GTE's Government Systems Group, Western Division. He provided direction in managing technical support facilities and methodology for all engineering activities.

Richard J. Sylvester

Dr. Richard Sylvester is Associate Technical Director in the Information Systems Division of The MITRE Corporation and Director of the MITRE Software Center. Previously, he was President and Chief Scientist of the Systems Productivity and Management Corporation which he founded in 1981. He was also a technical advisor on computer resources with the Aeronautical Systems Division of the U.S. Air Force at Wright Patterson Air Force Base, Ohio.

Dr. Sylvester has nearly 30 years of experience in software and acquisition programs encompassing both Army and Air Force weapons systems. He has been manager of the Mission Operations and Software Department at Martin Marietta in Denver and Director of New Jersey Operations at General Research Corporation.

Nelson H. Weiderman

Dr. Nelson Weiderman has been a member of the Computer Science faculty at the University of Rhode Island since 1971. From 1973 to 1983, he has served simultaneously as Director of their Academic Computer Center.

Since July 1985, he has been on leave from the university, and has an appointment as Visiting Senior Computer Scientist at the Software Engineering Institute (SEI) at Carnegie-Mellon University. At the SEI, Dr. Weiderman is the project leader of the Evaluation of Ada Environments project.

Charles A. Zraket

Charles Zraket is President and Chief Executive Officer of the MITRE Corporation. Mr. Zraket is responsible for MITRE's overall activities, including technical, administrative, and financial aspects on behalf of clients. Prior to this appointment, Mr. Zraket had been MITRE's executive vice president since 1978 and Bedford's general manager for the past year. He was senior vice president of technical operations from 1975 to 1978, taking on responsibility for all technical activities of the corporation.

Mr. Zraket joined MITRE at its founding in 1958, and has played a major technical and management role throughout the company's 28-year history. He has been a member of MITRE's Board of Trustees since 1978.

The illustration on page 21 was adapted from "Resource Analysis of Computer Program System Development," Alfred M. Pietrasanta, *On the Management of Computer Programming*, Auerbach Publishers, 1970.

Managing Editor: Joseph A. Sain

Designer: Debra J. Fiscus

Editors: Nancy J. Dashcund, Brian W. Donovan, Claudette G. Hanley, Margaret S. Jennings, Sarah A. Rolph

Copy Editor: Roberta A. Carrara

Phototypesetters: Barbara A. Vachon, Harold L. Xavier

Proofreaders: Harry Goodwin, Kendall H. MacInnis, Raymond W. Thuillier

Illustrator: Walter R. Osterberg

With thanks also to: Deborah L. Joy, Joan E. Lavery, Emily M. Morse, Dorothy B. Statkus, Jeanne M. Tourville, Joyce B. Wakefield

Printer: MITRE-Bedford Reproduction Services

1/87 1M A60351

MITRE

Sponsored by:
The Electronic Systems Division,
Air Force Systems Command
& The MITRE Corporation